THE REPRESENTATION OF ALGORITHMS

Robert M. Shapiro, et al

Applied Data Research, Incorporated
New York, New York

September 1969

RADC-TR- 69-313, Volume II
Final Technical Report
September 1969

THE REPRESENTATION OF ALGORITHMS

Applied Data Research, Incorporated

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, New York

102

THE REPRESENTATION OF ALGORITHMS

Robert M. Shapiro
Harry Saint

Applied Data Research, Incorporated

FOREWARD

This final technical report was prepared by Messrs. R. M. Shapiro, Harry Saint, R. E. Millstein, A. W. Holt, S. Warshall and L. Sempliner of Applied Data Research, Inc., Corporate Research Center, 450 Seventh Avenue, New York, N.Y. 10001, under Contract F30602-69-C-0034, Project 4594. Contractor's report number is CA-6908-2331.

The Rome Air Development Center project engineer was Miss Patricia Langendorf (EMIDD).

This report consists of two volumes:

Volume I: A Handbook on File Structuring

Volume II: The Representation of Algorithms

This technical report has been reviewed by the Office of Information (EMLS) and is releasable to the Clearinghouse for Federal Scientific and Technical Information.

This technical report has been reviewed and is approved:

Approved: Patricia M Langendorf
PATRICIA M. LANGENDORF
Project Engineer

Approved: A E Stoll
A. E. STOLL, Colonel, USAF
Chief, Intelligence and Recon Division

FOR THE COMMANDER: Irving J Gabelman
IRVING J. GABELMAN
Chief, Plans Office

ABSTRACT

The problem of representing mathematical processes is considered in the context of digital computer software and hardware.

TABLE OF CONTENTS

VOLUME I: A HANDBOOK ON FILE STRUCTURING

Page

Introduction 1

I. A Model of Cross-Indexing 7

II. Feature Cards 10

III. Edge-Notched Cards 13

IV. Indirect Coding 16

V. Superimposed Coding 24

VI. Combined Coding Techniques 29

VII. Is Retrieval Time a Linear Function of the Size of the Data Base? 30

VIII. The Volume of Cross-Indexing Information 32

IX. A Fundamental Difference between Item- and Descriptor-Organized Files 34

X. A Second Fundamental Difference between Item- and Descriptor-Organized Files 36

XI. Formulae for the Volume of Bits Transacted with 38

XII. Some Comments on the Volumetric Formulae. . 40

XIII. Computer Implementation of the Cross-Indexing Operation. 41

XIV. Computer-Implemented Item-Inverted File Organization 42

XV. Computer-Implemented Item-Sequenced File Organization 45

XVI. The Use of Indirect and Superimposed Coding in Computer Implementations 48

XVII. PDQ (Program for Descriptor Query) 49

XVIII. Batching or Buffering. 53

Page

XIX. Batching Queries and Updates 54

XX. An Important Asymmetry from the User's Point of View 56

XXI. An Alternative Method of Representing Lists in Inverted File Organizations . . . 57

XXII. An Analogous Alternative for Item-Sequenced File Organizations 60

XXIII. A Comparison of Three Organizations for Indexing 62

XXIV. A New Method for Performing List Intersections 69

XXV. Data Compression -- Another Encodement for Inverted Lists 76

XXVI. A Grammar for Defining Graph Representations of File Structures 79

XXVII. A Critique of Balanced Trees 98

XXVIII. Hashing and Secondary Storage 110

XXIX. Net Models -- Some Elementary Constructs . 133

XXX. A Model of Buffering 137

XXXI. A Model of Double Buffering 139

XXXII. Pipelined and Serial Phased Systems . . . 142

XXXIII. A Model of a Hardware Device -- The NCR CRAM Unit 148

XXXIV. A Highly Concurrent Net Model of the Cross-Indexing Grid 153

APPENDIX I. Petri Nets I-1

BIBLIOGRAPHY B-1

VOLUME II: THE REPRESENTATION OF ALGORITHMS
(separate book)

I. Introduction 1

II. Conventional Algorithmic Representations . 1

Page

III. Part-Part Matching 9

IV. Fundamental Restrictions Implicit in Conventional Representational Forms . . 16

V. Partial Ordering 23

VI. Variable-Names and Data Dependency Relations 31

VII. The Translation of Conventional Algorithms into Cyclic Partial Orderings 47

VIII. An Example of the Translation Procedure . . 60

IX. Pipelining 66

X. Control and Merges 70

XI. Proposed Extensions of the Representational Form 77

XII. Implications for Hardware Design 79

APPENDIX I. Petri Nets I-1

APPENDIX II. Warshall's Algorithm II-1

I. Introduction

In this report we intend to examine the problem of representing mathematical processes. We shall consider this problem in the context of digital computer software and hardware -- both because the availability of such computational machinery makes this the most useful avenue of approach and because this computational machinery has played such an important role in shaping the way in which people think about mathematical processes. In this context representations of mathematical processes are normally called algorithms. The word 'algorithm', however, tends to have a much narrower meaning, and the restrictions implied by the use of this word are built into the languages in which algorithms are commonly formulated. We shall begin by examining briefly the function of these standard representational forms. We shall try to determine exactly what representational restrictions they impose, and where these seem unjustifiable, we will propose alternative representational forms.

II. Conventional Algorithmic Representations

Let us consider a typical computing situation. A human being has some (perhaps relatively imprecise) notion of a mapping from some domain of inputs to some range of outputs; this mapping presumably takes form in his mind as a sequence of transformations on the inputs. He formulates this mapping precisely as an algorithm in some computer-oriented language like FORTRAN. A compiler then translates this definition of the mapping into a program which drives some computer in such a way that it performs the desired mapping. This procedure involves a series of translations -- from human notion to algorithmic language to hardware states. For these translations to be feasible there must be a reasonable similarity between the way in which human beings structure mappings, the structure of the algorithmic language, and the structure of the computing machinery.

The problem is that in designing languages to express algorithms (and computers to perform them), we have two -- often conflicting -- aims. The first of these aims is to provide human beings with the most convenient representational medium possible for the definition of mappings. The second is to provide a representational form which can be conveniently translated into the most efficient hardware implementation possible with respect to space and time (i.e., how much equipment is required for how long).

With respect to the first aim, a number of criticisms can be made of algorithmic languages. For purposes of this discussion, however, we shall assume at the outset that, at least for a large and interesting class of problems, these languages -- particularly with respect to their fundamental conceptual organization -- provide the most convenient possible representational medium for the definition of input/output mappings by human beings. We will concern ourselves instead with the second function of algorithmic languages -- that of providing a satisfactory source representation for the translation into the most efficient possible hardware implementation. We shall argue that from this point of view the fundamental conceptual view of mathematical processes which underlies standard algorithmic languages (and machine design) is unsatisfactory. We shall propose a representational form with a different conceptual groundwork and demonstrate the feasibility of translation from standard algorithmic languages into this representational form. We shall try to indicate both how this representational form might enable us to exploit current computing machinery more efficiently and, more importantly, what implications it might have for the design and exploitation of more powerful machinery.

We shall begin by examining in some detail the view of mathematical processes which provides the foundation for algorithmic languages and machine design. Let us consider, for example, a flowblock diagram of an algorithm defined in some language like FORTRAN. The diagram is a directed graph whose nodes are the flowblocks; each flowblock contains a totally ordered set of FORTRAN statements. The flowblocks are connected by directed arcs; each arc is an output of exactly one flowblock, and an input to exactly one flowblock. Cycles and loops are permitted. Each flowblock has at least one input arc and at least one output arc with the exception of a unique flowblock called entry, which has no input arc, and a unique flowblock called exit, which has no output arc. Since this diagram is to be a representation of a process, it is meaningless without some sort of simulation rule. This is provided by creating an entity called control. Control can be thought of as a unique token which moves through the diagram in discrete steps, residing at any given time at exactly one statement. To begin simulation of the algorithm, control is placed on the first statement of the entry flowblock. Within a flowblock, control moves from one statement to its immediate successor (the statements within a flowblock are always totally ordered); from the last statement of a flowblock, control may move along any one of the output arcs to the first statement of some other (or the same) flowblock. Each time control resides at a statement, that statement is executed exactly once. When control arrives at the last statement of the exit flowblock, the simulation is completed. We now have a rough picture of an algorithm functioning: a unique entity named control wanders through a "flow diagram" bringing to life one statement at a time as it drifts by. The two most interesting features of this picture are (1) that at any given time during a simulation, control resides at exactly one statement and (2) that in the course of one simulation, control may visit the same statement many times.

We must now examine the individual statements. In order to avoid unnecessary complications, let us invent a simplified version of FORTRAN which permits:

(1) two types of I/O statements: the word 'READ' followed by exactly one variable-name, and the word 'WRITE' followed by exactly one variable-name;
(2) assignment statements, consisting of exactly one variable-name followed by '=' followed by either one variable-name (or one integer) or else by two variable-names (or two integers, or one variable-name and one integer) separated by an arithmetic or Boolean operator;
(3) control statements of two types: 'GO TO' followed by a statement-name, and 'IF' followed by a variable-name

followed by three statement-names.

Let us look first at a typical assignment statement: A=B+C. The variable-names (A , B , and C in this example) act as "placeholders" for values. We could translate this statement as follows: add the value currently assigned to B and the value currently assigned to C ; assign the result to A . Hence, we call A the result and B and C the operands. Once control encounters this statement, A will continue to "stand for" the value assigned to it by the execution of the statement until control encounters another assignment to A (or re-encounters the same assignment to A). In other words, any variable-name occurring on the right side of an assignment statement (i.e., as an operand) represents the result of the most recently executed assignment to that variable-name. Because the same variable-name may be designated as the result in several different statements, and because control may pass to the same statement more than once, a given variable-name may represent many different values during one performance of the algorithm. However, the fact that control can reside at only one statement at a time guarantees that at any given time during the performance of an algorithm, a given variable-name represents (at most) one value (since there can be at most one most-recently-encountered assignment to that variable-name). Note that the concepts control and variable are interdependent.

Initially, at least, we can think of I/O statements as "incomplete" assignment statements. A READ statement assigns a value to a variable-name; a WRITE statement uses a variable-name as an operand.

Control statements do not directly affect the values of variables. Instead they determine the path of control from one flowblock to another. In particular an IF statement uses the current value of some variable (which we may think of as the operand of the IF statement) as the criterion for determining which of several alternative "paths" (i.e., output arcs from the flowblock) control will take. Consequently, such statements are commonly called "decisions".

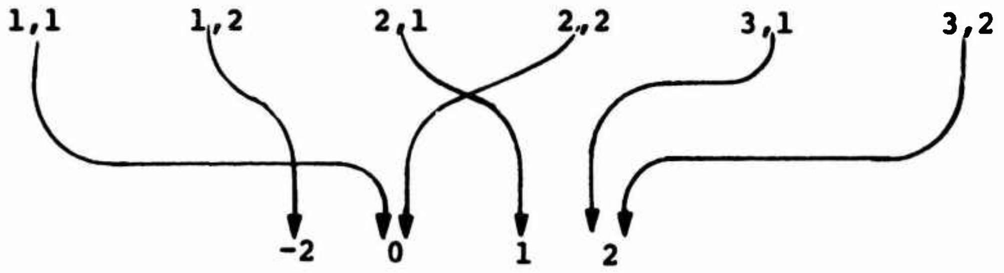
We now have a conceptually complete model of an algorithm. If we eliminate the two-dimensional aspects of this picture by arranging the statements in a list, we have a typical algorithm definition in an algorithmic language. One statement in the list can be designated the initial statement and another the terminal statement. To "run" the algorithm we place our "control token" on the initial statement. Control then moves down the list executing one statement at a time; some of the statements may be control statements whose execution may cause control to be sent to some statement other than the immediately subsequent statement; when control arrives at the terminal statement, the "run" is completed. Hardware performance of an algorithm is pictured with the same conceptual machinery. A program consists of a set of instructions, each of which occupies one location in memory. Memory is totally ordered -- each location has a unique successor. There is a control counter which (roughly speaking) always contains the memory address of the next instruction to be executed and there is some sort of central processor which executes the instructions one at a time. Each time an instruction is executed, the address in the control counter is incremented so that it contains the address of the next location in memory. The execution of a transfer instruction, however, may place the address of some other memory location in the control counter. An instruction which specifies the execution of some arithmetic or logical operation may designate a memory location whose contents are to be used as an operand; or an instruction may designate a memory location in which the result of some such operation is to be stored. Thus memory locations may be used very naturally in a way analogous to variable-names in an algorithmic language representation. In general, the translation from algorithmic language to hardware will be relatively straightforward.

III. Part-Part Matching

This conceptual machinery for representing mathematical processes is in some respects extremely powerful. The fact that during one performance of an algorithm the same statement may be executed many times and that the same instance of a variable-name may represent a different value each time, means that a relatively small set of statements may represent an arbitrarily long sequence of different computations. In hardware terms this means that a relatively small computing device can be programmed to perform a relatively long and varied sequence of operations. Let us illustrate more clearly, with the help of a simple example, exactly how this conciseness is achieved. Consider the following algorithm consisting of five statements:

```
READ A
READ B
A=A-B
A=A*B
WRITE A
```

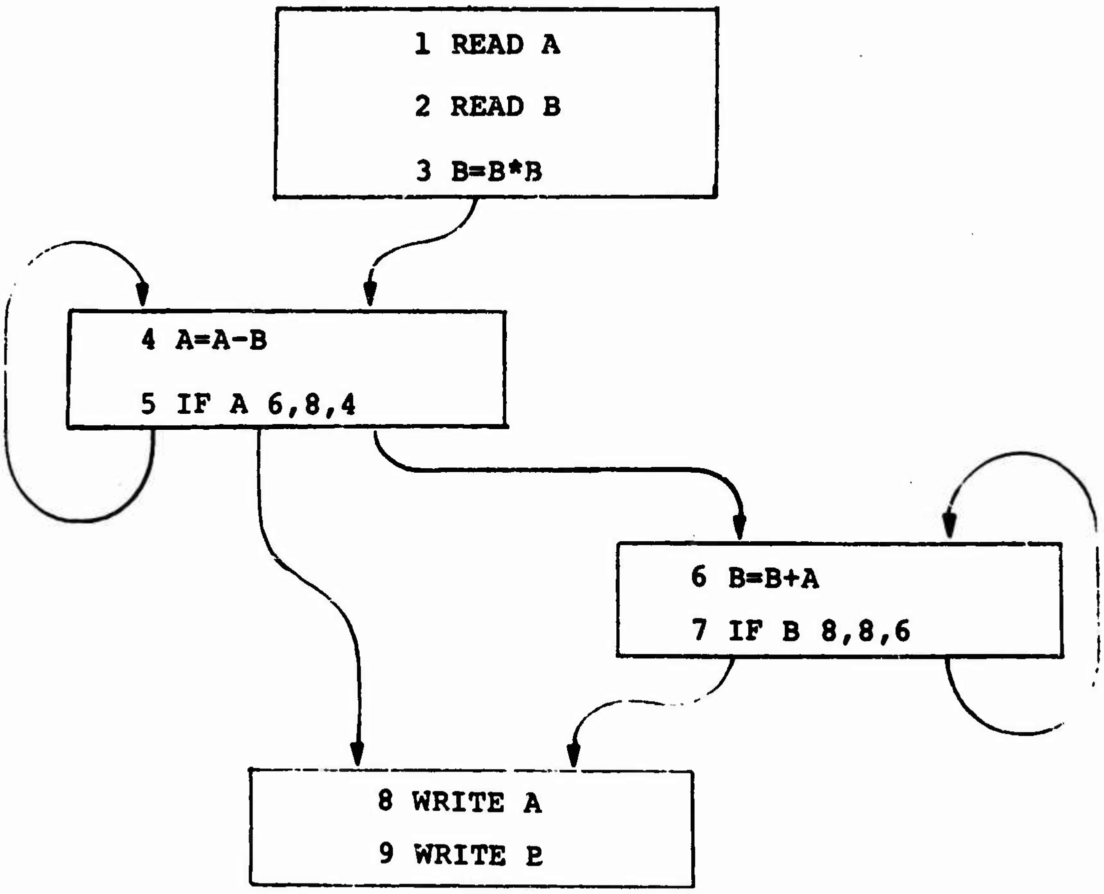
Suppose that the input domain is defined as follows: the possible input values of A are the integers 1, 2, and 3; the possible input values of B are the integers 1 and 2. The algorithm could then be thought of as representing the following mapping:



However, the algorithm defines the mapping as a sequence of arithmetic transformations on an ordered input pair so that from this point of view we may think of the algorithm as representing a set of computational histories -- one for each input pair in the domain, as follows:

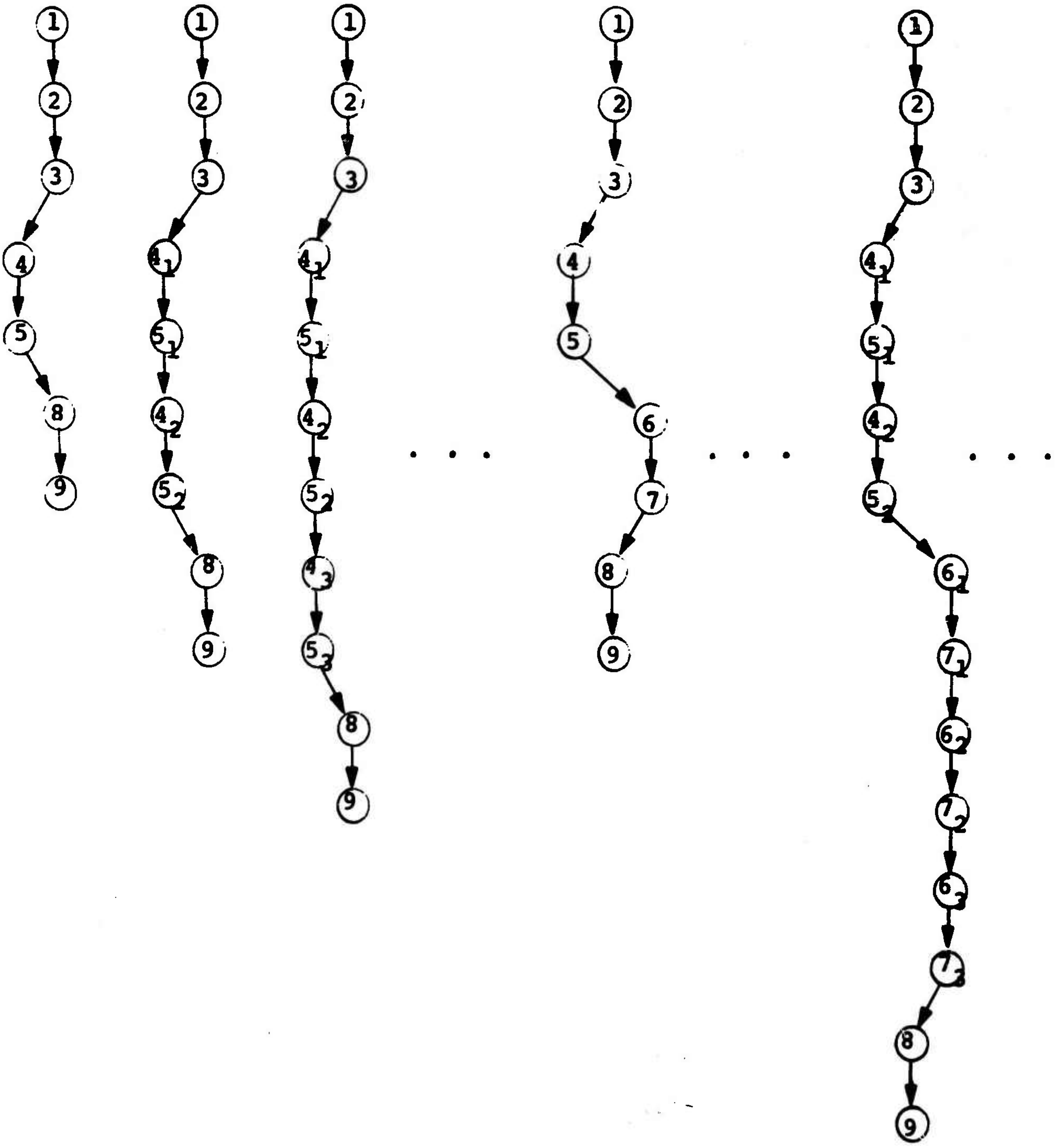
INPUT 1,1	INPUT 1,2	INPUT 2,1	INPUT 2,2	INPUT 3,1	INPUT 3,2
0=1-1	-1=1-2	1=2-1	0=2-2	2=3-1	1=3-2
0=0*1	-2=-1*2	1=1*1	0=0*2	2=2*1	2=1*2
OUTPUT 0	OUTPUT -2	OUTPUT 1	OUTPUT 0	OUTPUT 2	OUTPUT 2

The algorithmic definition provides <u>one</u> representation for *all* of these computational histories by what we will call *part-part matching*: "parts" of different computational histories are "matched" to each other in such a way that all the computational histories may be "overlaid" and given one representation. Each variable name then represents a *set of mutually exclusive values* (for example, in any given performance of the algorithm, B represents either 1 or 2), and each statement therefore represents some set of mutually exclusive computations (for example, A-B represents either 1-1 or 1-2 or 2-1, etc.,). Let us next examine the role of control with the help of a slightly more complicated example:



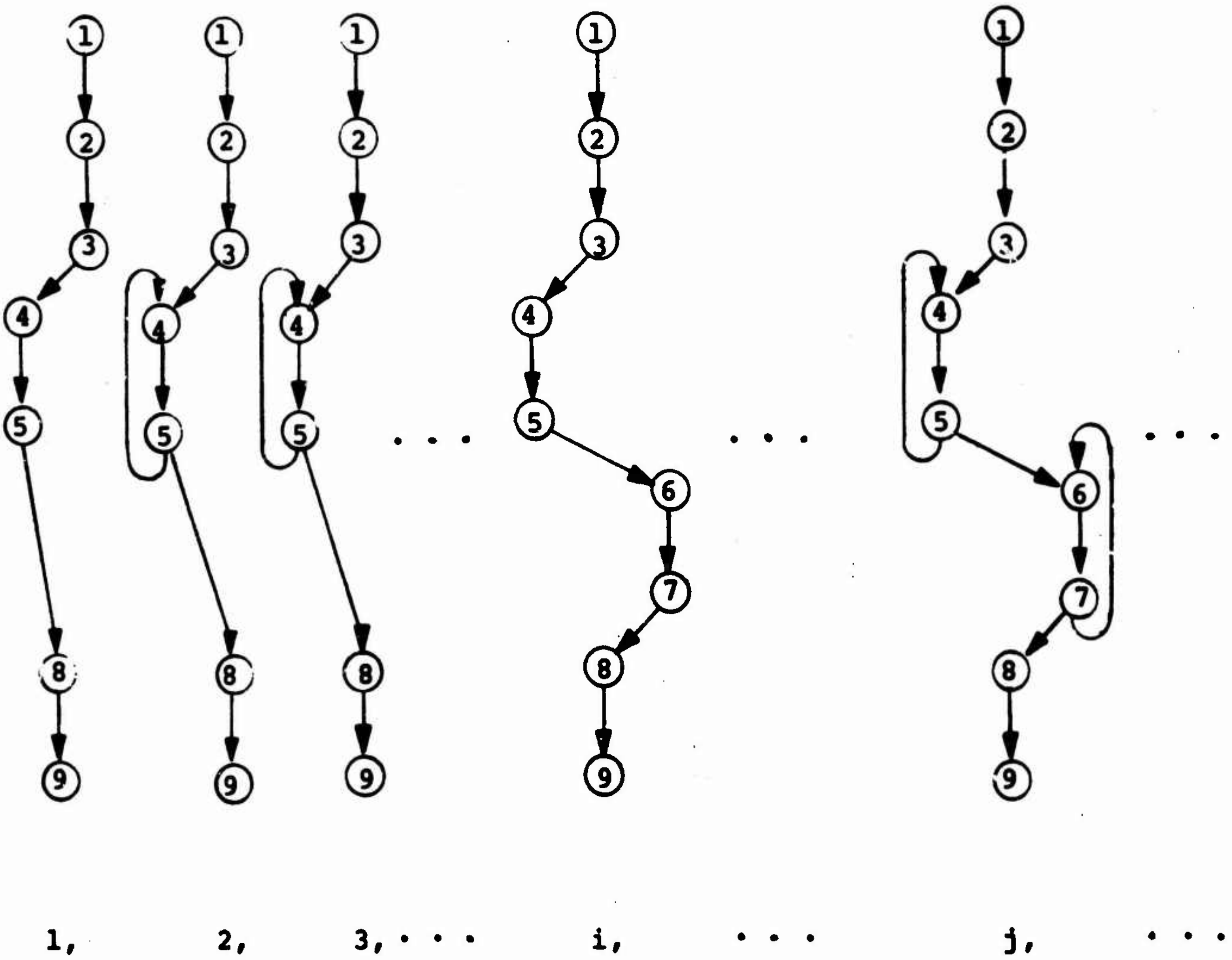
```
1 READ A
2 READ B
3 B=B*B

4 A=A-B
5 IF A 6,8,4

6 B=B+A
7 IF B 8,8,6

8 WRITE A
9 WRITE B
```

We will now take for granted the kind of part-part matching discussed in the preceding paragraphs and consider only the range of "control histories" which this algorithm represents. By "control histories" we mean the set of possible paths from entry to exit. (Each "control history" may of course represent many computational histories.) Given some finite domain of input values, we could in principle represent each possible control history as a directed graph as follows: (see next page)

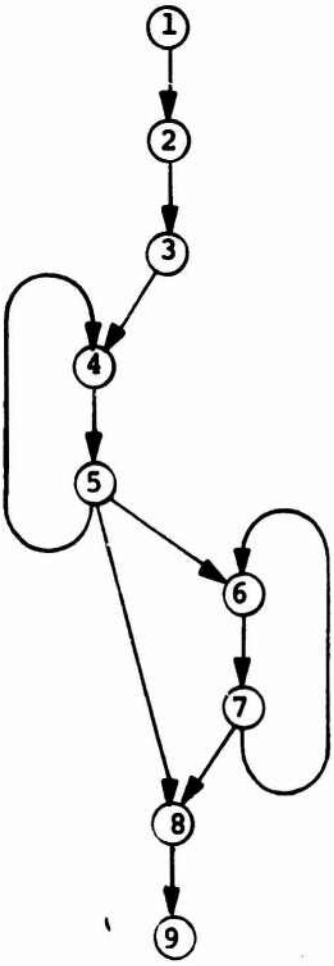

1
2
3
4
5
8
9
1
2
3
4_1
5_1
4_2
5_2
8
9
1
2
3
4_1
5_1
4_2
5_2
4_3
5_3
8
9
. . .
1
2
3
4
5
6
7
8
9
. . .
1
2
3
4_1
5_1
4_2
5_2
6_1
7_1
6_2
7_2
6_3
7_3
8
9
. . .

1, 2, 3, . . . i, . . . j, . . .

If we took such a set of all possible histories as our starting point, we could view the task of producing an algorithmic language definition of the algorithm as a process of further part-part matching: we first "fold up" each of the individual histories by matching different parts of the same history to each other, as follows:



In history j , for example, we have matched operation 5_1 with operation 5_2; since operation 5_1 had operation 4_1 as its successor and operation 5_2 had operation 6_1 as its successor, in the folded-up representation of the history operation 5 has both operation 4 and operation 6 as successors. We can now "overlay" all the folded up histories by similarly matching parts of different histories to each other. We match, for example, operation 5 in history 2 with operation 5 in history j . The result of this part-part matching is of course equivalent to a flow diagram of an algorithmic language definition of the algorithm.

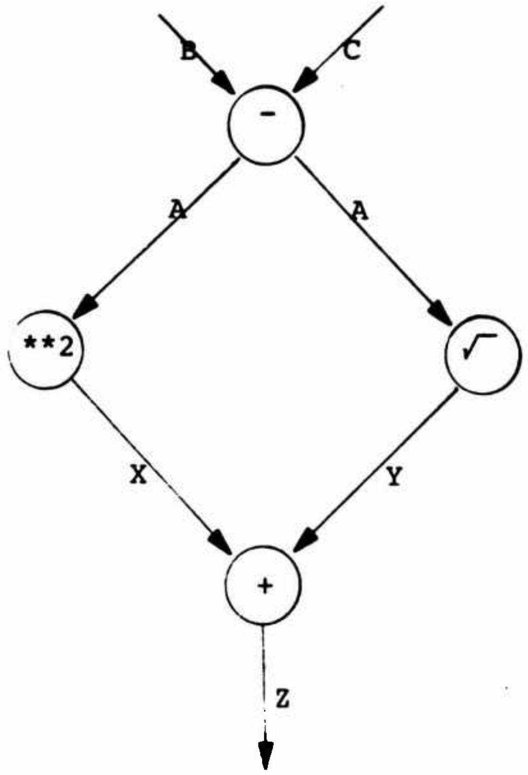


IV. Fundamental Restrictions Implicit in Conventional Representational Forms

Clearly, the part-part matching information implicit in standard algorithmic definitions is extremely useful, and we will want to retain it in any alternative representational form we might propose. Specifically, this information leads to a very efficient use of space. Roughly speaking, standard algorithmic formulations approach maximum efficiency with respect to space and minimum efficiency with respect to time. They make very inefficient use of time because, very simply, they require that only one thing be done at a time. The notion of control as a unique "entity" which passes from one statement or instruction to another forces a total ordering on all the computations in the performance of an algorithm; every history will necessarily consist of a totally ordered sequence of operations. Consider the following two sequences of statements:

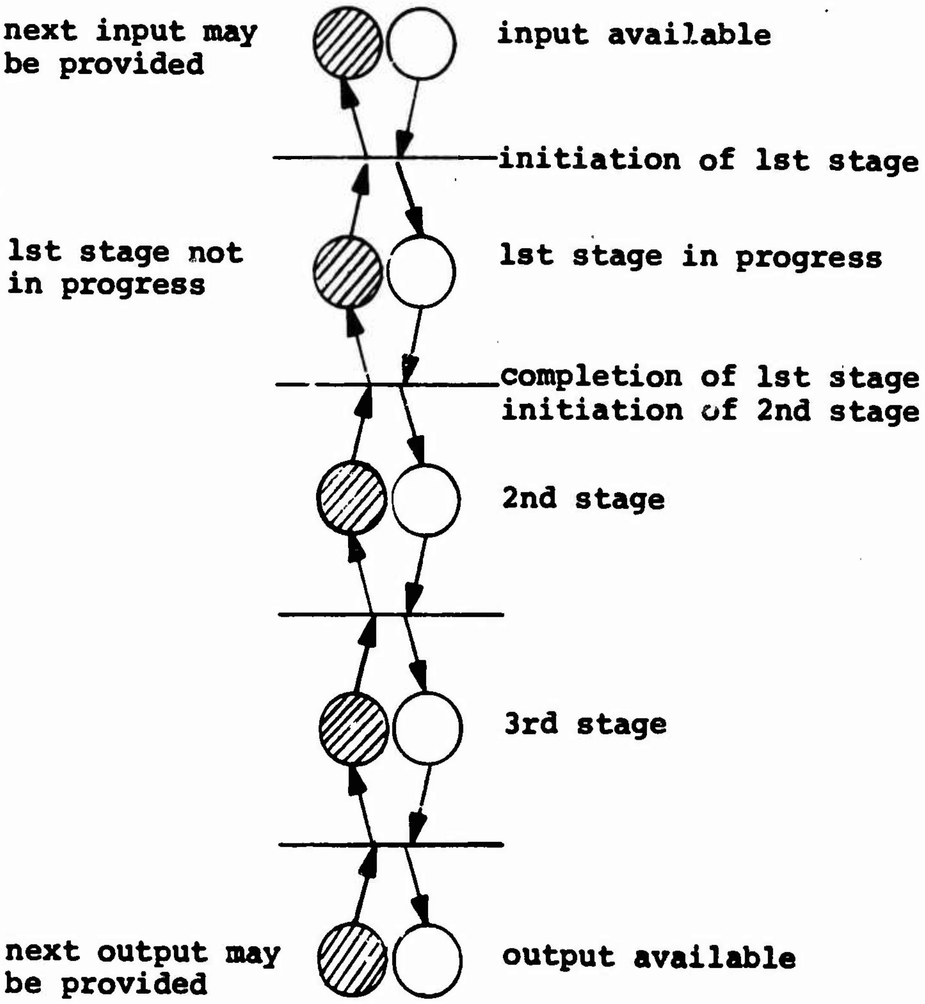
$$\begin{array}{ll}
A=B-C & A=B-C \\
X=A^2 & Y=\sqrt{A} \\
Y=\sqrt{A} & X=A^2 \\
Z=X+Y & Z=X+Y
\end{array}$$

These two sequences are computationally equivalent; an algorithm writer would nevertheless be forced to choose one of them arbitrarily. Assuming we had adequate computing machinery, however, we could make more efficient use of time by performing the second and third statements concurrently. We might then find it useful to exhibit this possibility explicitly by representing these four statements as a partial ordering defined by the data dependencies.



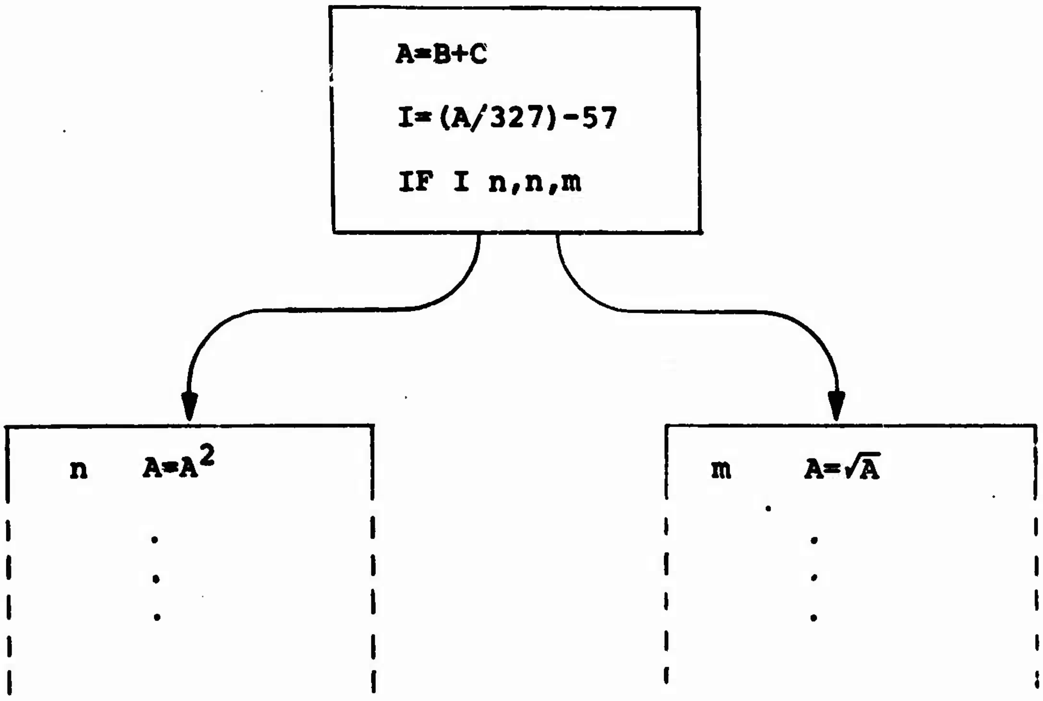
In general an algorithm consists of a set of partially ordered operations, where the partial ordering is determined by the data dependencies. We can obviously increase efficiency with respect to time by performing unordered operations concurrently.

Furthermore, once we admit the possibility of exploiting partial ordering information by performing different operations concurrently, it emerges that other arbitrary restrictions have been implicitly imposed by the standard representational forms. Consider, for example, the following net model of a three-stage process:



The three stages are totally ordered, which would seem to exclude any possibility of concurrent operation. Let us assume that each stage takes one unit of time; it will therefore take three units of time for each mapping of an input to an output. On the other hand, if we further assume that inputs can be provided (and outputs accepted) by the environment at the rate of one per time unit, then as soon as the first stage has finished with the first input, it may accept the next input. Thus the inputs can be pipelined so that all three stages are operating concurrently. The throughput rate would then approach one output per time unit. If we could abandon the notion of control we might similarly be able to represent the possibility of pipelining in an algorithmic context. Suppose, for example, that the net diagram above is a model of a program loop, with the process stages representing the individual instructions and the ordering relations corresponding to the data dependencies within the loop. Then, despite the fact that the instruction executions (within any one iteration of the loop) must be totally ordered, all the instructions may be performed concurrently (by overlapping executions from successive iterations) so that the throughput rate is limited only by the duration of the most time-consuming single instruction. Similarly we could represent the possibility of pipelining the algorithm as a whole -- that is, the possibility of an algorithm or program working on more than one input set concurrently.

Another interesting restriction implicit in the standard view of algorithms is that of doing only what must necessarily be done. Consider the following example:



If the computation of I is extremely lengthy we might profitably "defer the decision". That is, while we are computing I and then "making the decision", we might concurrently pursue both of the alternative branches -- even though one of them will turn out to have been "unnecessary". We could then use the result of the decision to choose which of the alternative values of A we wished to retain.

¶ Given the possibility of concurrent operation, we might also wish to question the automatic one-one mapping of variable names to equipment locations. Two uses of the same variable name might be entirely unrelated in terms of data dependency and thus potentially concurrent if mapped to different equipment locations. Other types of space/time trade-offs might also become more interesting. A computation which is a "bottleneck" in the performance of an algorithm might be "duplicated" in a hardware representation.

Many such possibilities for taking advantage of partial ordering information and potential concurrent operation are already exploited to a limited extent on both the hardware and software levels. At the level of individual instruction execution there is normally a very high degree of concurrent operation, of course. However, at what we might call the "programmable level" of machine operation there is very little. Machines like the CDC 6600, the CDC 7600, and the IBM 360/91 permit some concurrent execution of instructions. The CDC 7600 furthermore, has functional units which may be pipelined (so that a given functional unit may be working on several instructions at a time). Programs for these machines, however, must consist of totally ordered sets of instructions, and the central processor decodes the instructions sequentially; furthermore the register and functional unit reservation schemes impose further restrictions on parallel operation. Consequently, potential concurrency is exploited to a very limited degree and only very locally. Many machines allow concurrent I/O processing, and there are a number of machine designs which allow several central processors to pursue loosely related computational paths concurrently. The 360/91 allows a kind of decision-deferral or "look-ahead" which involves pursuing the "most likely" branch provisionally before a conditional branch instruction has actually been executed.

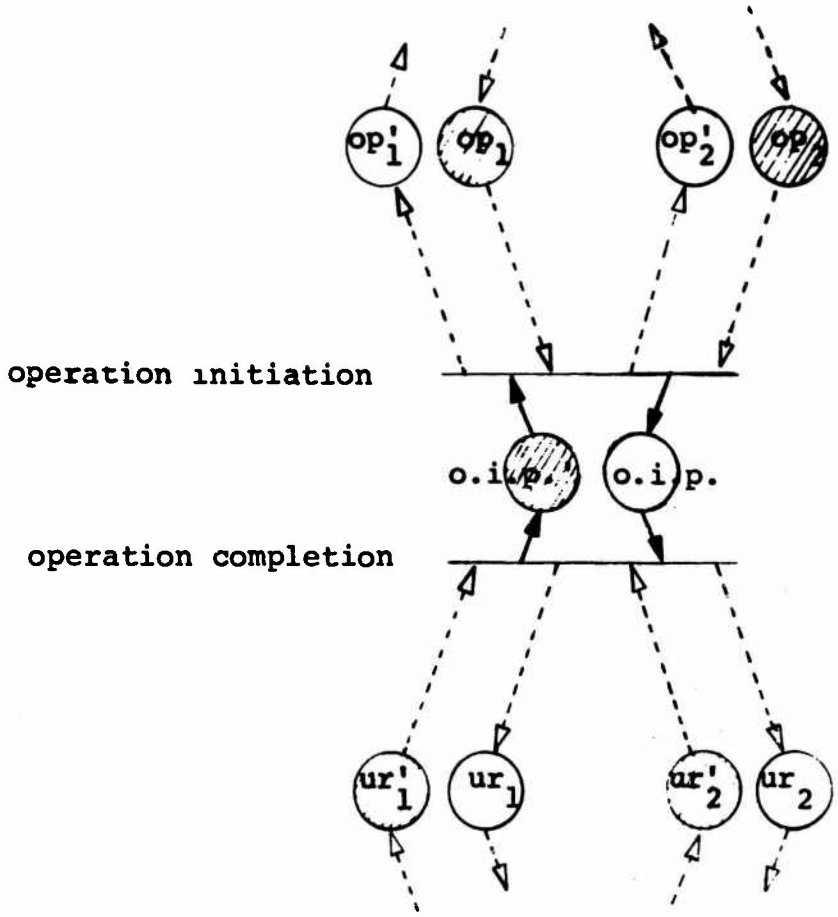
Frequently, even though machine operation is sequential, one sequence of operations will be more efficient than another, computationally equivalent sequence (for example, because it requires less intermediate storage) so that, on the software level, there are a number of optimization techniques which use partial ordering information for resequencing. The same kind of information may also expose redundant computations -- caused either by several redundant expressions or by an expression which is invariant within a loop. Again, these techniques are applied independently and (except for recognition of invariance within a loop) only locally -- usually within a single flow-block. All of these procedures, both hardware and software, are closely related: they are piecemeal attempts to provide more efficient hardware implementations by circumventing arbitrary restrictions imposed by the representational forms in which algorithms are defined. No consistent global exploitation of these possibilities can be achieved, however, because the necessary information is inaccessible in such representations.

V. Partial Ordering

Many optimization techniques involve translations of algorithms (or more usually, segments of algorithms) into partial orderings representing data dependencies. However, such partial ordering techniques normally preclude statements of the form " a precedes a "; consequently cycles must be excluded. Furthermore, no attempt is made to represent the interaction of decisions with data dependencies. This means that generation of all partial ordering information for an algorithm would involve producing a partial ordering for each possible control history of the algorithm. This problem remains serious even if we restrict our attention to one program loop. Either we must limit ourselves to one iteration of the loop -- in which case we exclude all information about concurrencies across different iterations of the loop (it is precisely this information which can provide us with "pipeline" solutions) -- or we must "unwind" the loop (i.e., treat it as one long "straight-line" sequence of computations rather than as a loop), which, although it does lead to explicit representation of concurrencies across successive iterations, necessarily means throwing away the part-part matching information.

We should like to be able to represent algorithms as partial orderings of data dependencies without sacrificing useful part-part matching information. We will therefore use Petri nets as our basic representational medium. Petri nets can be used to exhibit explicitly both partial ordering and part-part matching information because they represent the behavior of cyclic systems of partially ordered events and states. For a brief description of Petri nets see Appendix I.

Let us first consider the use of Petri nets to model algorithms in which control is as straightforward as possible -- that is, algorithms without any conditional branches and hence with only one possible control history. We can represent each arithmetic or logical operation with the following schema.



o.i.p. : operation in progress

o.i.p.': operation completed, not in progress

op_1 : operand$_1$ available

op_2 : operand$_2$ available

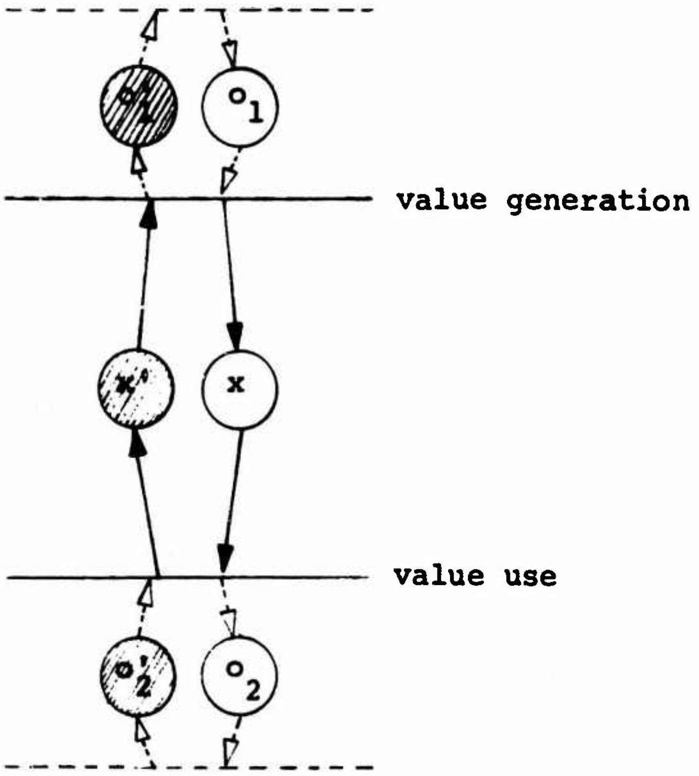
op'_1 : operand$_1$ used in operation, not available

op'_2 : operand$_2$ used in operation, not available

ur'_1 : result of operation may be made available; use$_1$ of previous result (as an operand for some operation) has already taken place

ur_1 : result of operation available for use$_1$

Note that each use of a given result is represented uniquely. Accordingly, each use of a variable-name as an operand (i.e., each occurrence of the variable-name on the right side of an assignment statement) will be represented as follows:



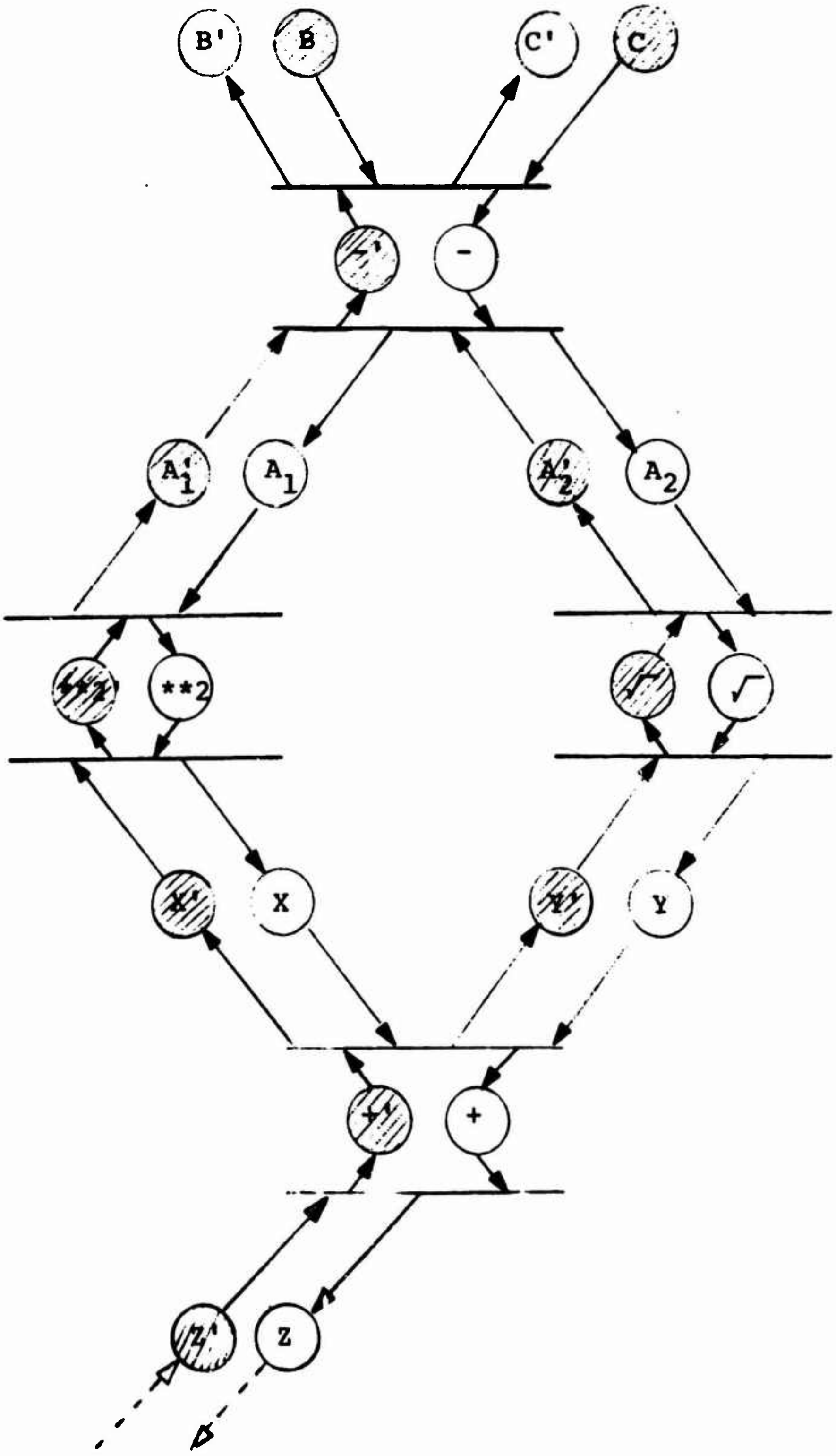
o_1 : operation$_1$, which generates values for x, in progress

o_2 : operation$_2$, which uses x as an operand, in progress

x : x is available as an operand for operation$_2$. The value of x may not yet be changed.

x' : x used as an operand for operation$_2$. The value of x may be changed as a result of operation$_1$.

We would then represent the four line example which we used earlier as follows:



$A=B-C$

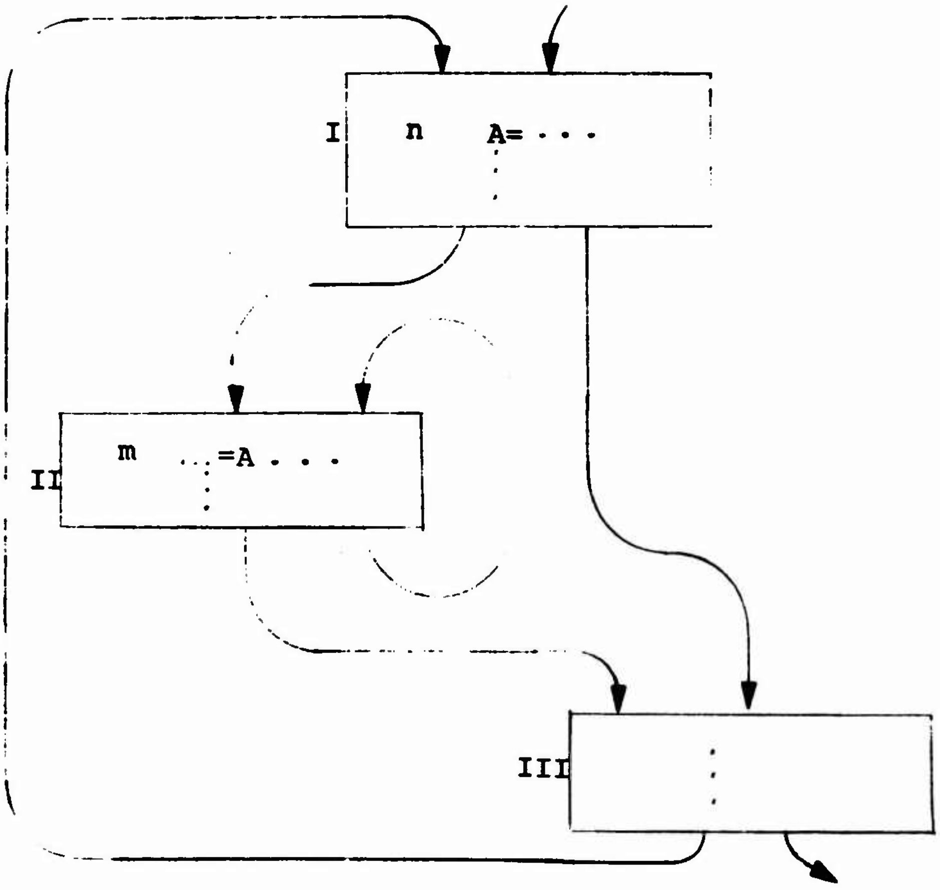
$X=A^2$

$Y=\sqrt{A}$

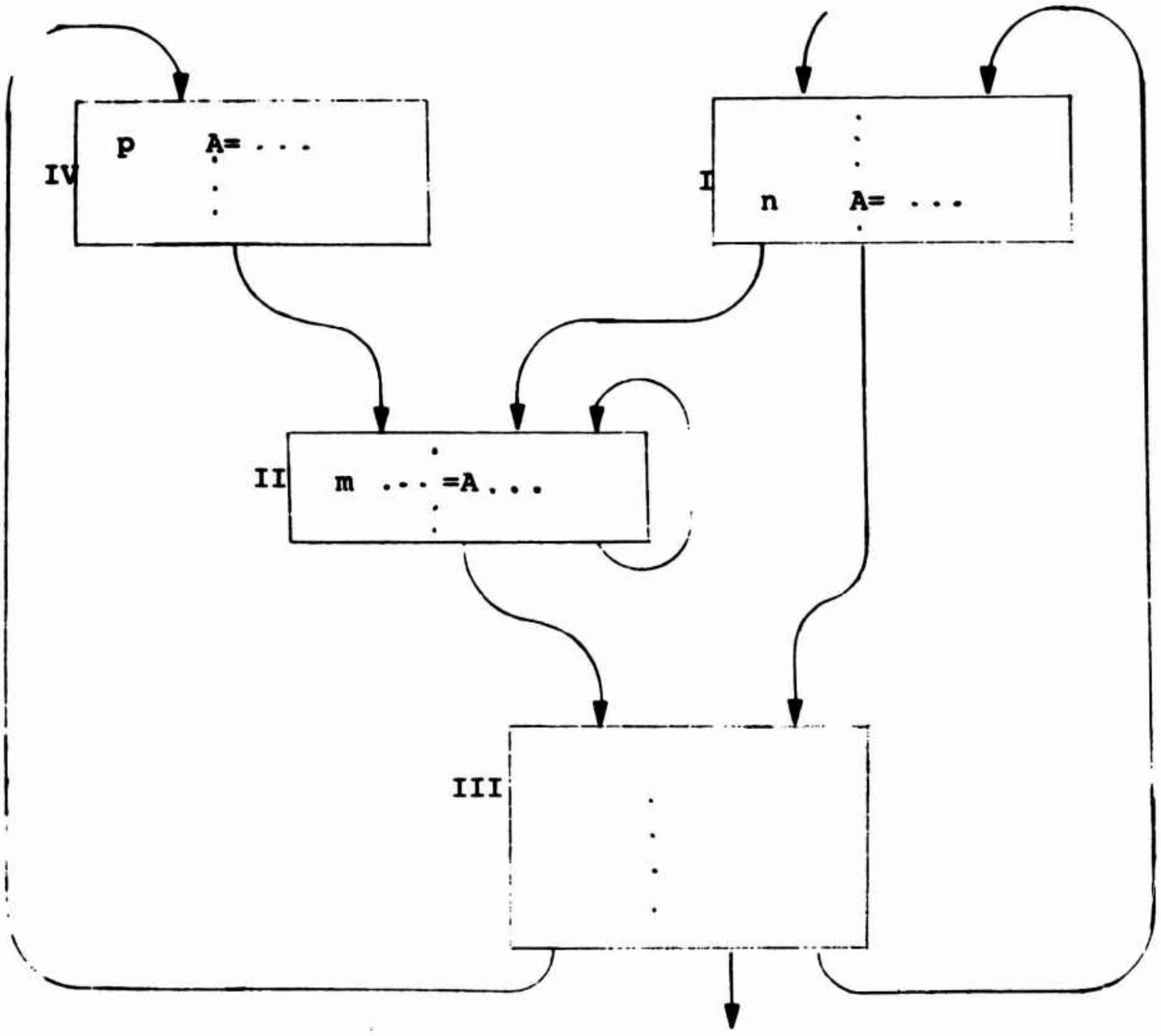
$Z=X+Y$

Note that the schemata for operations and for variable-uses have cyclic behaviors and that therefore the algorithmic representation which we have constructed from them also behaves cyclically. One important consequence of this is that our representation expresses not only "forward" data dependencies (the computation of the next value for X cannot begin until the current operand value for A has been computed) but "backward" data dependencies as well (the value of A may not be changed to its next value until the current value has been used by the operations which compute X and Y).

As long as an algorithm contained no decisions, we could apply these schemata throughout to obtain an adequate representation. As soon as we admit branching, however, this procedure is no longer adequate, because decisions render the dependency relations variable. Consider, for example, the following flowblock diagram, in which we will be concerned only with the variable A :



Statement n generates a value for A which is used in statement m . Each time a value is generated for A at statement n (each time control flows through flowblock I), that value may be used at statement m once, or many times, or not at all (control may flow from I to II to III; or it may flow from I to II and then recirculate through II any number of times; or it may flow from I to III and back to I again). Or we might complicate the picture slightly so that there are two alternative statements, either one of which may have generated the value used in any given execution of statement m .

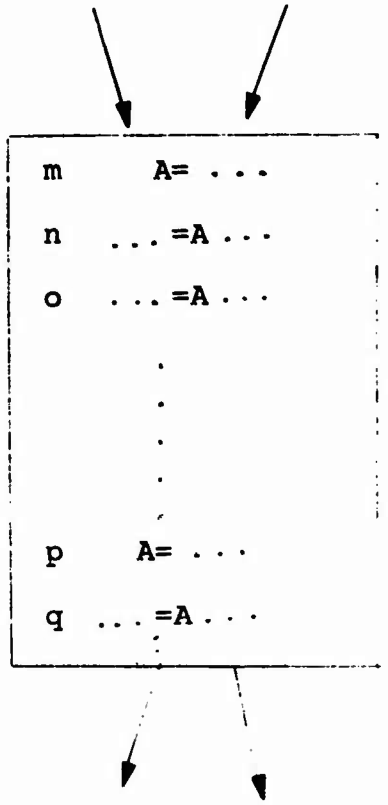


In any algorithm which contains branching, the data dependencies are "variable" -- that is, they are determined by the particular path "chosen by control" when the algorithm is executed. It should be kept in mind, furthermore, that both forward and backward data dependencies are at issue; we are interested not only in when the apprcpriate value for an operand is available and may be used but also in when a value is no longer needed and a new value may be provided. The data dependencies in an algorithm with branching constitute what we might call a "variable partial ordering". Clearly, before we can consider the problem of representing such variable partial orderings, we will have to deal with the problem of extracting the necessary data dependency information from the algorithmic language definition of an algorithm.

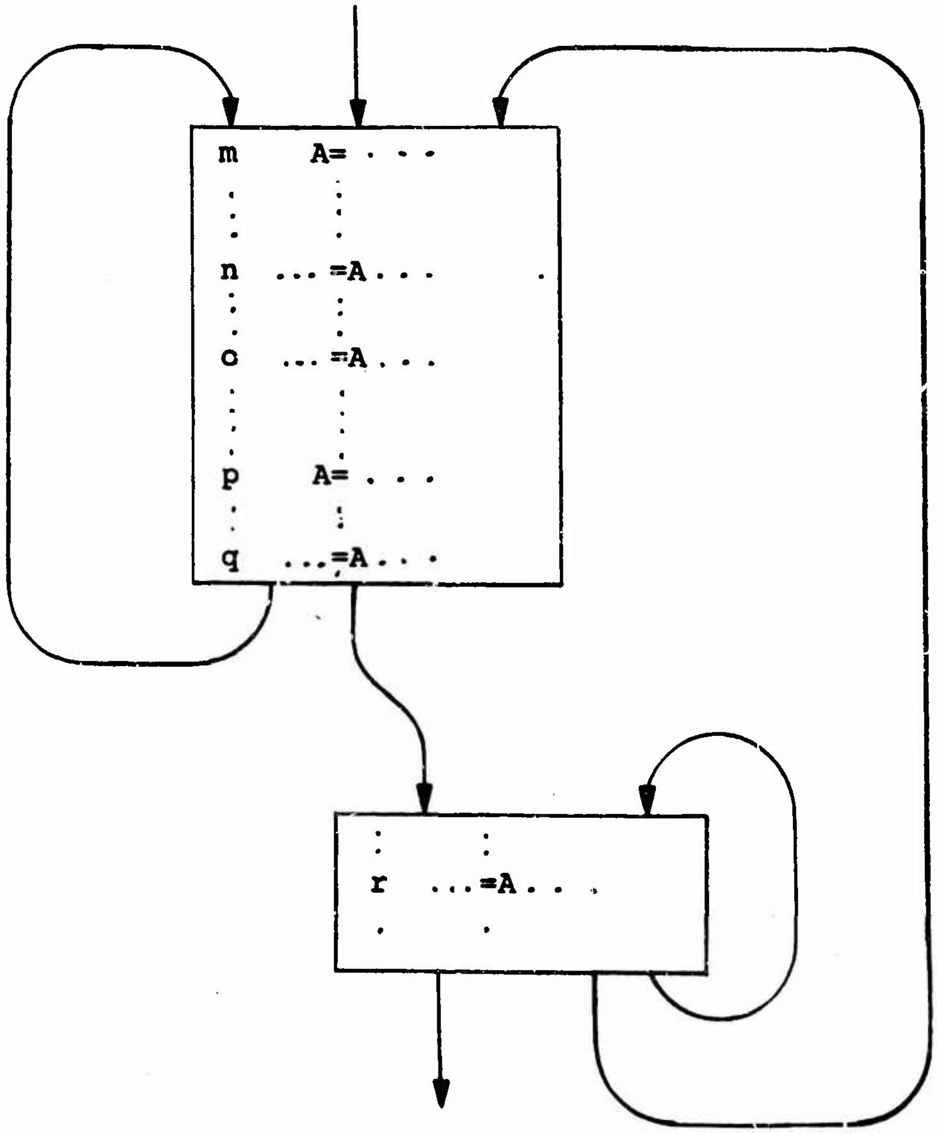
VI. Variable-Names and Data Dependency Relations

We have already discussed one role of variable-names: a variable-name represents a set of mutually exclusive values; at any time during performance of the algorithm, (at most) one of these values will hold. We have also considered another role of variable-names: different (and possibly unrelated) uses of the same variable-name may be -- and normally will be -- mapped to the same machine location; thus the various uses of a given variable-name constitute part-part matching information. (Of which we may or may not wish to take advantage -- as we have already pointed out, it will frequently prove advantageous to map different uses of the same variable-name to different machine components in order to allow these uses to be concurrent.) We will now want to consider another role of variable-names: we will want to examine the ways in which variable-names interact with control to determine data dependencies.

¶ Consider the following example, in which, again, we are interested only in the variable A .

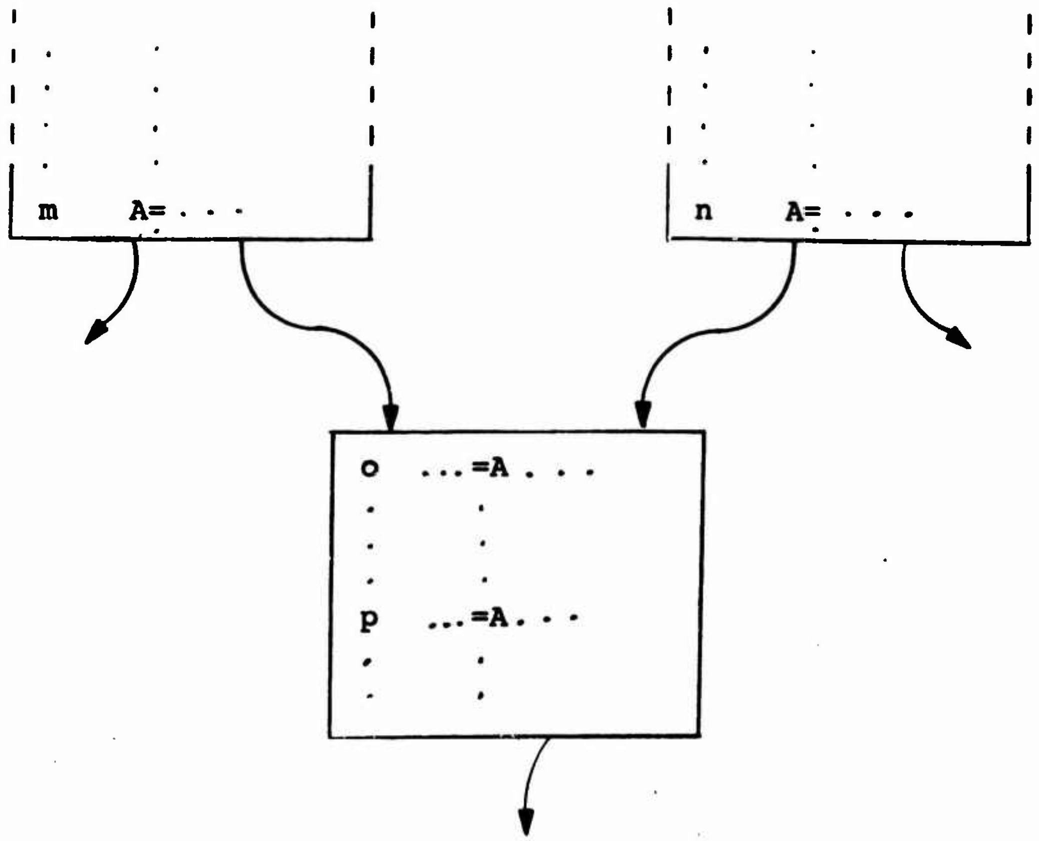


We said earlier that an occurrence of a variable-name on the right side of an assignment statement (i.e., a use of a variable) represents the result of the most recently executed assignment to that variable-name. Whenever statement n or statement o is executed, the occurrences of A in these statements must necessarily represent the value produced by the most recent execution of statement m . Consequently statements n and o are ordered with respect to statement m : statement n is later than statement m , and statement *o* is later than statement *m* . Because the uses of *A* in statements *n* and *o* always represent the value generated by the most recent execution of statement *m* (i.e., *m* is the *only* assignment statement which can produce values for those uses), we call those uses of *A* members of the *A* equivalence class generated by statement *m* . Similarly, the use of *A* in statement *q* is a member of the *A* equivalence class generated by statement *p* . Let us now expand the example as follows.



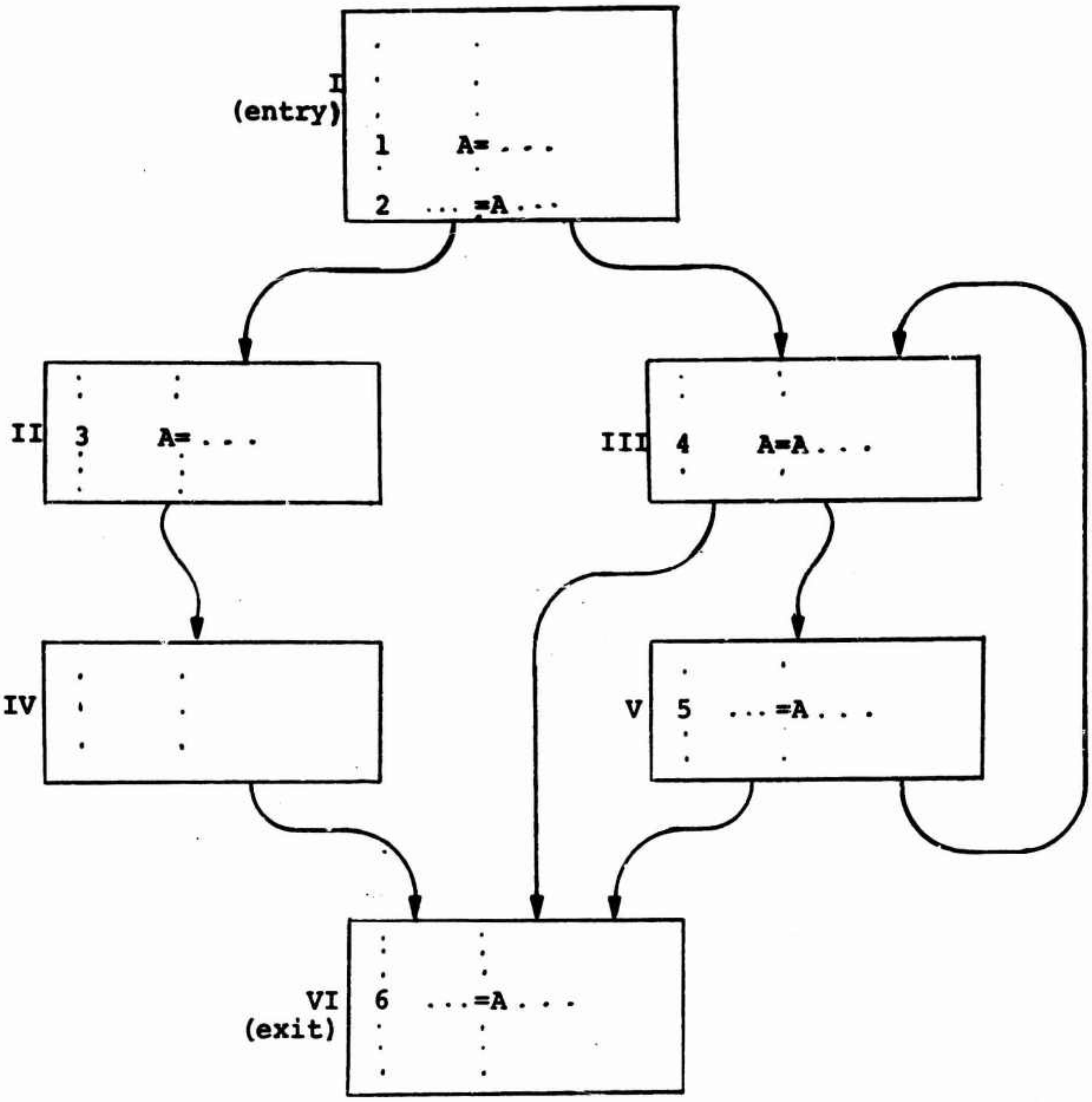
After each generation of the A equivalence class generated by statement p , the use of A in statement r may occur once, or many times, or not at all before the generation of some other A equivalence class. However, the use of A in statement r always (if and whenever it occurs) represents the value generated by the most recent execution of statement p , and therefore it is a member of the A equivalence class generated by statement p . One way, then, in which variable-names and control interact to determine data dependency is in the generation by assignment statements of equivalence classes, which define ordering relations between operations. Each occurrence of a variable-name on the left side of an assignment statement represents the generation of an equivalence class; each use of that variable-name for which that assignment statement is *always* the most recent assignment to that variable-name is a member of that equivalence class.

Let us now consider an example which illustrates another type of data dependency.



Here there are two possible statements, m and n , which may generate a value for the use of A in statement o . These uses of the variable-name A express both a part-part matching and a set of ordering relations. The results of two alternative computations are "merged"; whenever statement o is executed, it uses as an operand the most recent result of either statement m or statement n -- whichever was executed most recently. In the context of our earlier discussion, we can describe merges as the result of part-part matching -- either the "folding up" of one history or the "overlaying" of different histories (or both). In hardware terms, such part-part matching, represented by merges, allows the mapping of alternative control histories onto the same equipment. Statements m and n both generate A equivalence classes. But since the use of A in statement o represents a value which may have been produced by either m or n , it cannot be a member of either equivalence class. Because we will want the A equivalence classes to constitute a partition of all uses of A , we will let the uses of A in statements o and p constitute a third equivalence class. Roughly speaking, we can think of this equivalence class as having been generated when either statement m or statement n has been executed, and a decision has been made to go to flowblock III. In the flowblock diagram we would locate the generation of the equivalence class at the "merge-point" -- that is, at the entry to flowblock III. Note that whenever statement o is executed, this merge-point will then always be the most recent point at which an A equivalence class was generated. In terms of the data dependencies, a set of alternative ordering relations has been defined: either o is later than m, or o is later than n . We have, then, two types of equivalence classes, which represent ordering relations between operations. We shall want to partition the uses of each variable-name into equivalence classes. To accomplish this we use an algorithm by Warshall, which is described in Appendix II. Here we shall restrict ourselves to a very brief account of the algorithm. It might be useful to recast the problem of partitioning variable-uses into equivalence classes in more familiar terms. A common optimization problem -- and one which is normally dealt with only locally -- is the elimination of redundant computation, or common subexpression elimination. Suppose, for example, that the expression SINF(A) appears twice in an algorithm. We would like to know whether we can compute the sine of A once for both uses. We would like to know, roughly speaking, if both instances of A "always represent the same value." More precisely, is there a point p in the flow diagram such that on no path from p to either of the uses is there an A-assignment and such that there is no path from any A-assignment to either of the uses which does not pass through p ? (If such a point exists, we can safely place the computation of the sine of A there.) Based on our definition of an equivalence class, we can restate this question as follows: Are both instances of A members of the same equivalence class? Warshall's algorithm, then, may be thought of as a global solution of the problem of common subexpression elimination.

As an example, let us take the flow diagram below and apply Warshall's algorithm to the variable A . Only the occurrences of A actually appear in the diagram, and the statements in which they occur have been numbered for convenience.

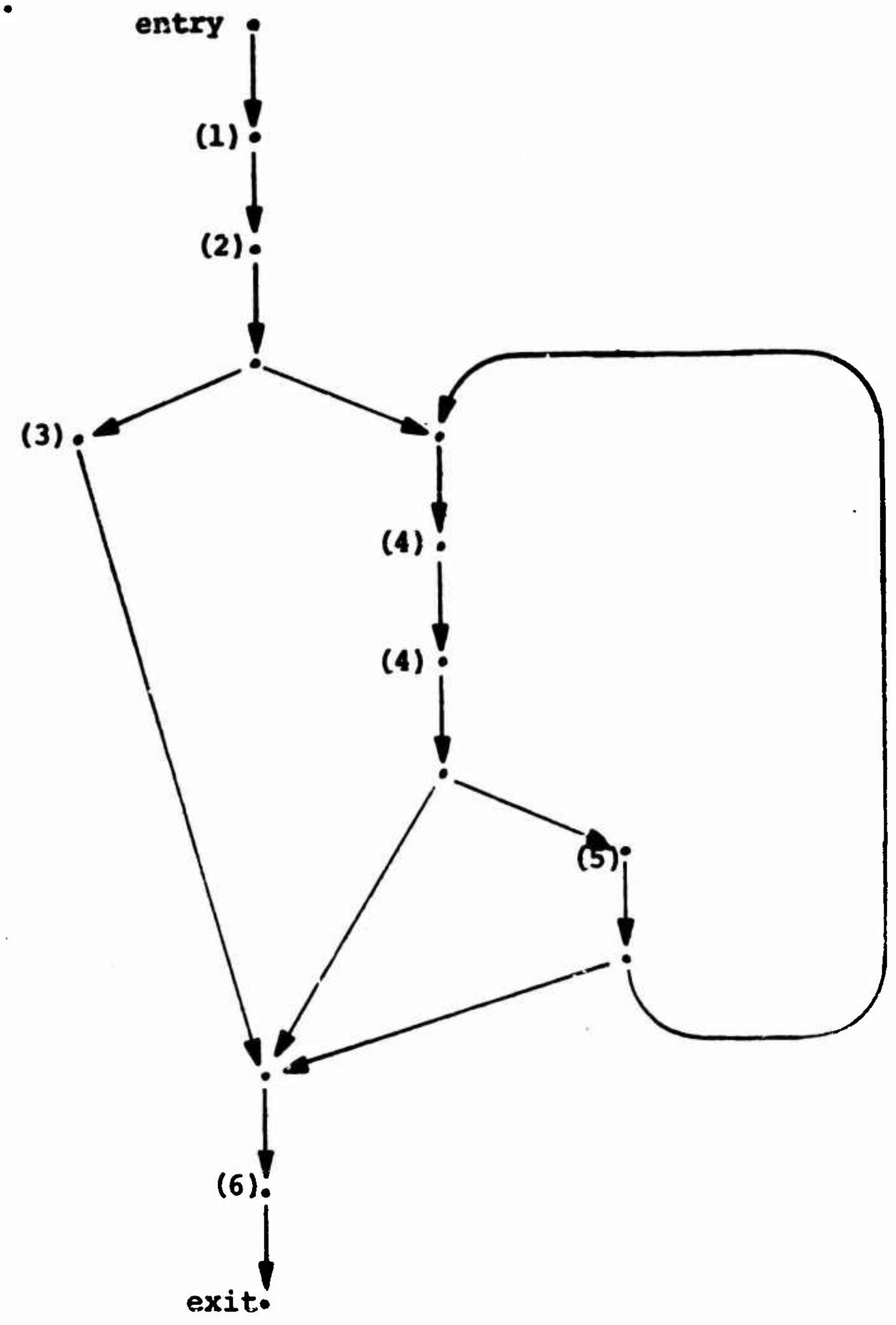


We expand this graph by replacing each flowblock-node F with a totally ordered set of nodes Φ , as follows:

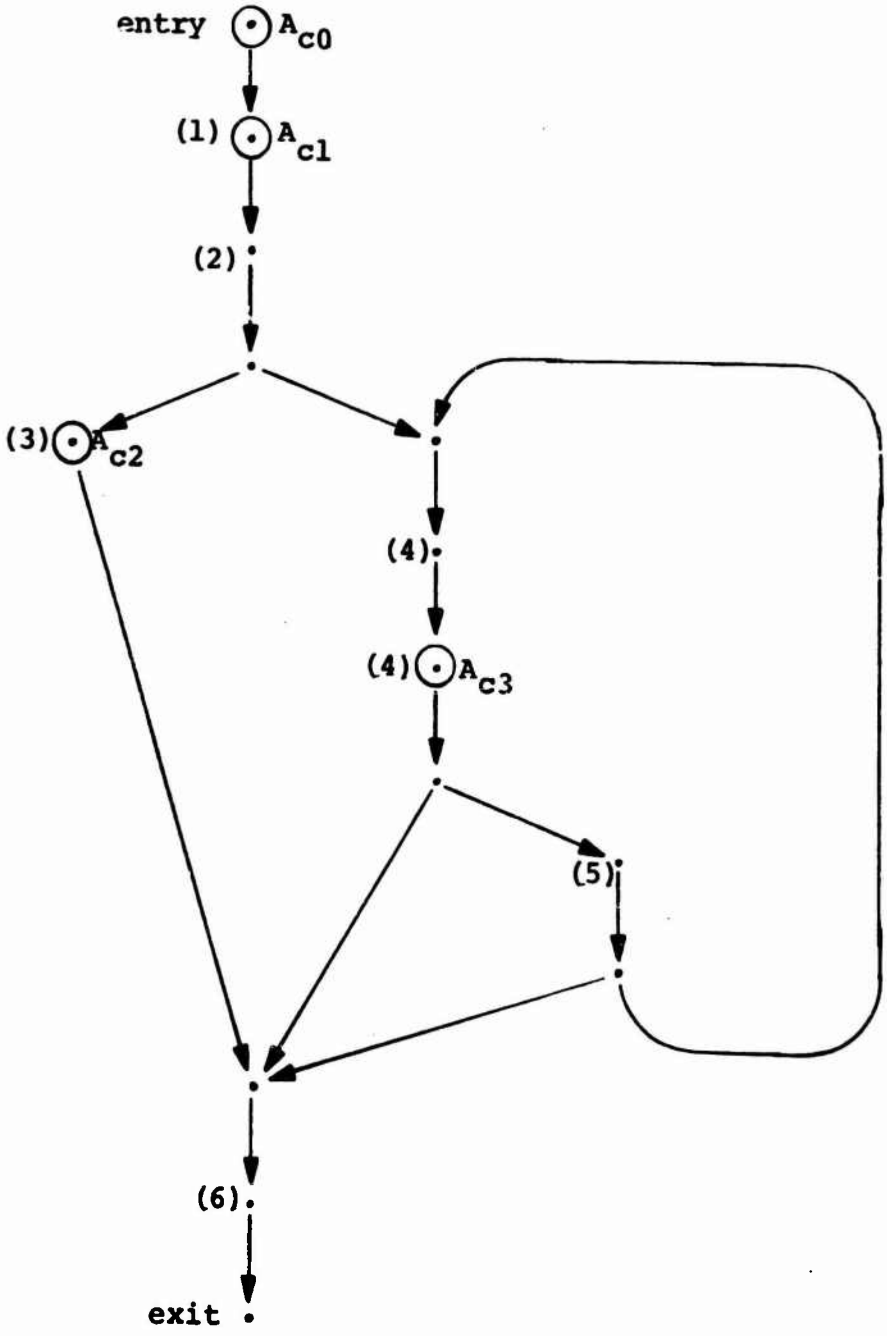
- if F is the entry flowblock, one node called the entry-node, which is the earliest node in Φ ; or

- if F has more than one input arc, one node called a flow-node, which is the earliest node in Φ ; and
- a set of instance-nodes, one corresponding to each instance of A in flowblock F ; these nodes are ordered according to the order in which the corresponding instances of A occur within the flowblock (where the left side of an assignment statement is later than the right side); and
- if F has more than one output arc, one node called a decision-node, which is the latest node in Φ ; or
- if F is the exit flowblock, one node called the exit-node, which is the latest node in Φ .

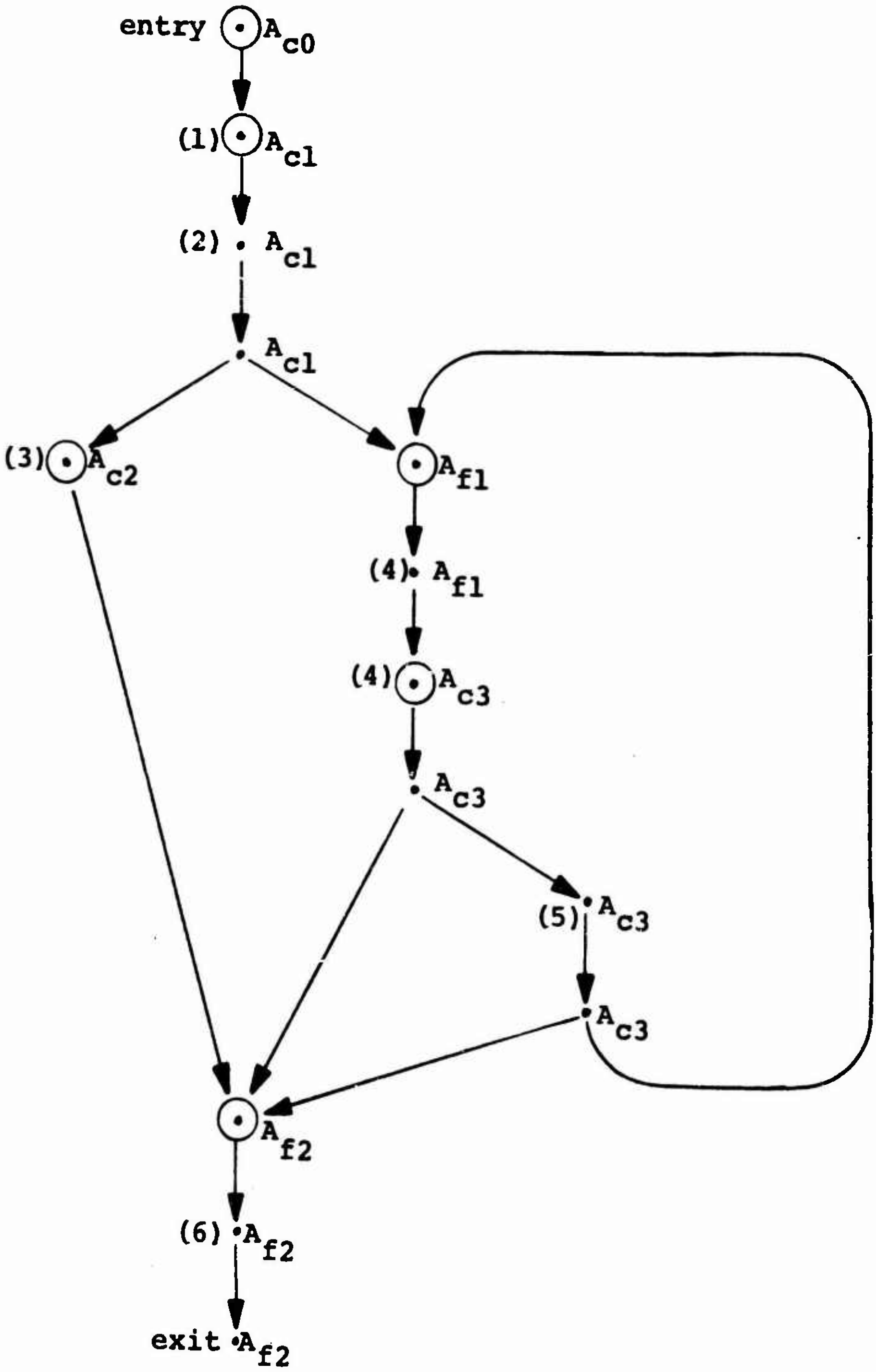
All input arcs of F become input arcs of the earliest node in Φ ; all output arcs of F become output arcs of the latest node in Φ . If Φ is empty (i.e., F has one input arc, one output arc, and contains no instance of the variable), then it must have some unique precedessor flowblock G and some unique immediate successor flowblock H ; replace the arc from G to F and the arc from F to H with one arc from G to H . The resulting graph for the example above is the following:



Note that there is still a unique node which is earlier than every other node in the graph (the entry-node) and a unique node which is later than every other node in the graph (the exit-node). The purpose of the algorithm is to subscript the instances of A in such a way that they are partitioned into equivalence classes. This means that we want to identify a minimal set of nodes in the above graph as equivalence-class-generators such that every other node in the graph has a unique most recent equivalence-class-generating ancestor. We know that all assignments to A generate equivalence classes; therefore, we will circle all nodes representing left-side instances of A and label them uniquely as A_{c1} , A_{c2} , ... , A_{cn} . We will also circle the entry node and label it A_{c0} .

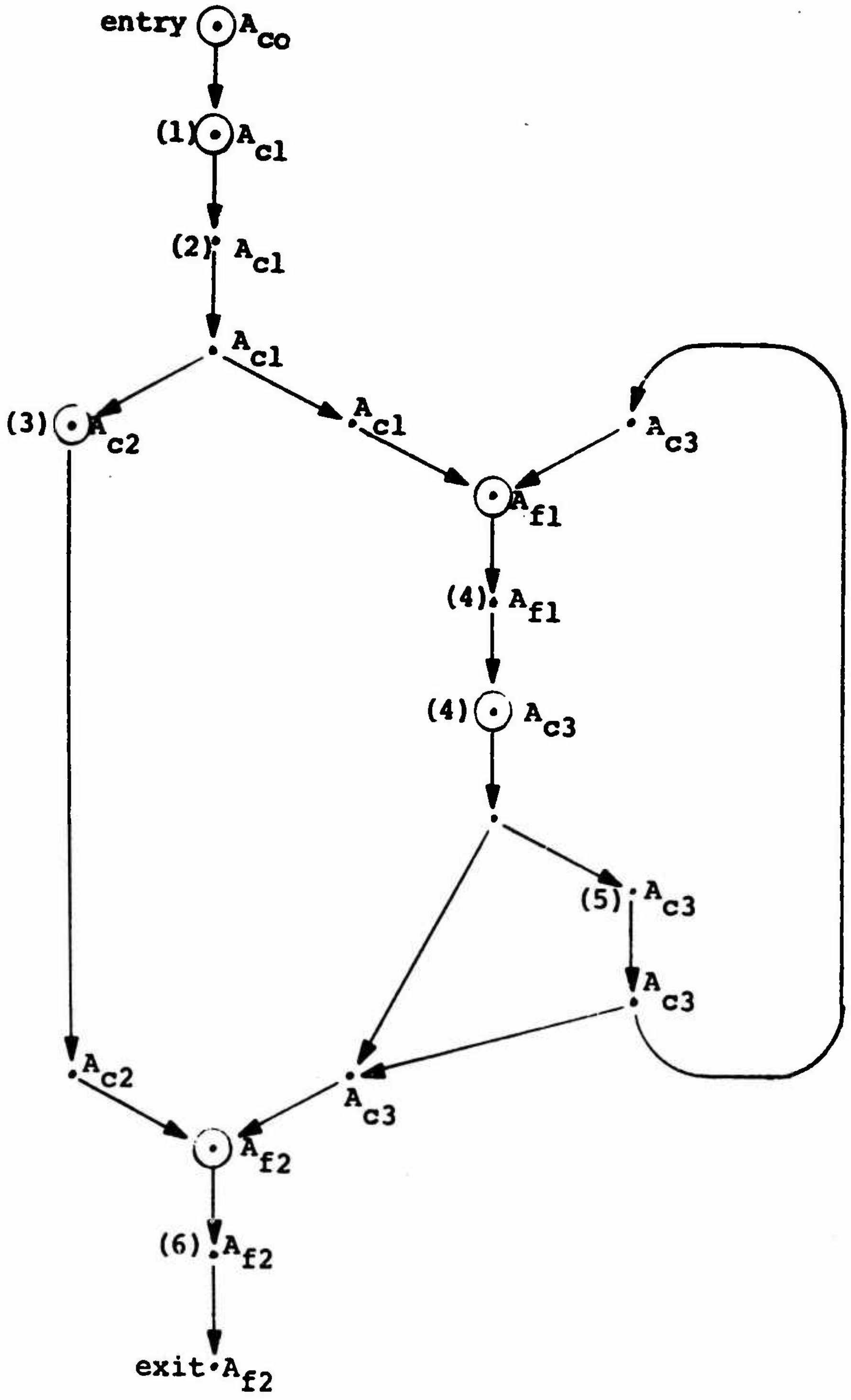


It remains to determine the equivalence classes which must be generated because of merges. Roughly speaking, the algorithm accomplishes this by pushing the name of each circled node along directed arcs to all uncircled nodes which can be reached without encountering another circled node. When two different names meet on an uncircled node, that node is circled; such newly circled nodes are uniquely labeled as A_{f1} , A_{f2} , ... , A_{fm} . The names of the newly circled nodes are also propagated until no more nodes may be circled. (It should be intuitively clear that only flow-nodes are candidates for circling). Upon completion of the algorithm every node is either circled or has associated with itself the name of exactly one circled node: namely, its unique most recent circled ancestor. The names define the partition into equivalence classes. Each circled node corresponds to an equivalence-class-generating event -- either an assignment to $\underline{A}$ or a merge.

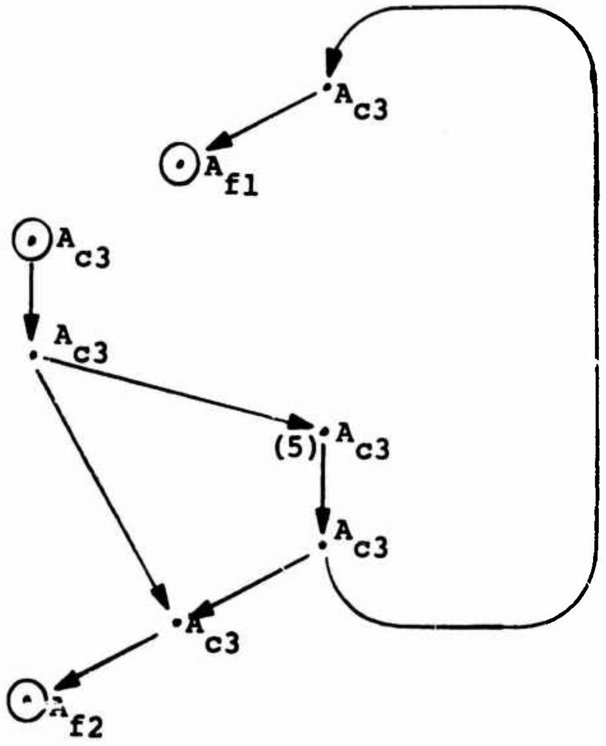


Millstein and Warshall prove that the solution is unique and minimal.

¶ Let us make one further modification in the above graph, as follows. For each circled flow-node, which defines some equivalence class A_{fi} , consider the names associated with those nodes which are its immediate predecessors. For each such name A_{kj} (where $\underline{k}$ represents either $\underline{f}$ or $\underline{c}$) -- and there must be at least two different names -- introduce a new uncircled node into the graph, and assign it the name A_{kj} . Introduce new arcs such that this new node is the immediate successor of each A_{kj} node which was an immediate predecessor of the circled A_{fi} node. Eliminate the arcs from these immediate predecessors to the circled A_{fi} node, and introduce a new arc from the new A_{kj} node to the circled A_{fi} node. Application of this procedure to the example above produces the following graph:

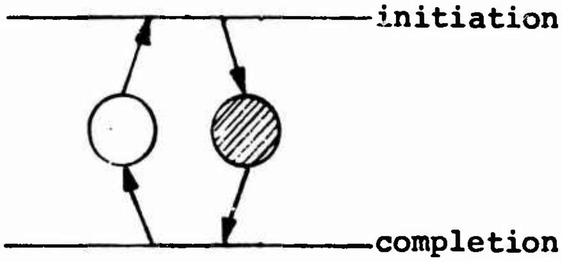


We call this graph <u>the complete p-graph of A</u> . Let us provide several further definitions for future use. We call the set of all nodes in the complete p-graph of $\underline{A}$ which have associated with them the name A_{ki} the <u>members of the equivalence class A_{ki}</u>. We call the set of all nodes which are immediate successors of members of some equivalence class A_{ki} but which are not themselves members of the equivalence class A_{ki} <u>the exit nodes of A_{ki}</u>. We define <u>the graph of the equivalence class A_{ki}</u> as the subgraph of the complete p-graph of $\underline{A}$ which contains the members of A_{ki} together with the exit nodes of A_{ki}. Thus the graph of the equivalence class A_{c3} in our example would be the following:

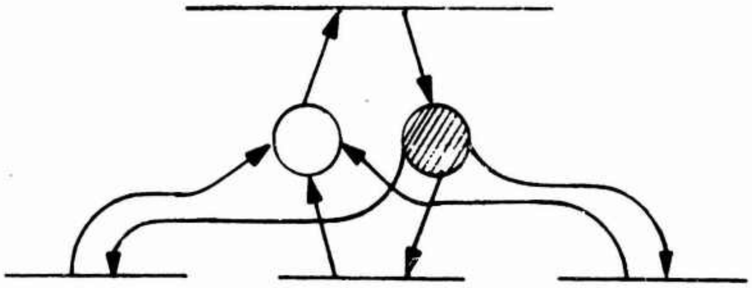


VII. The Translation of Conventional Algorithms into Cyclic Partial Orderings

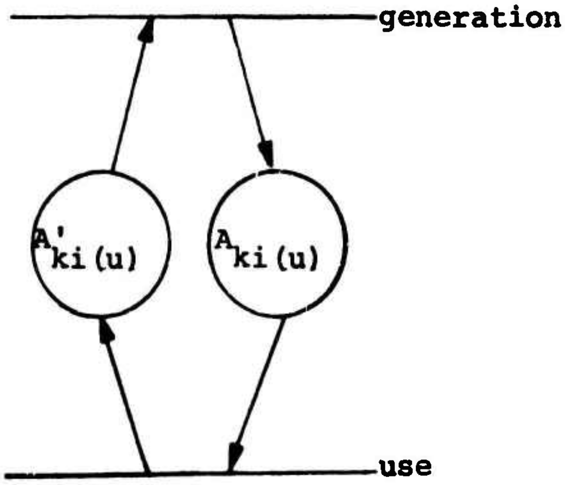
Let us assume that we have applied Warshall's algorithm to each variable in some algorithm. We can now consider the problem of giving this data dependency information explicit representation and of relating it to decisions. Let us continue to represent operations as before.



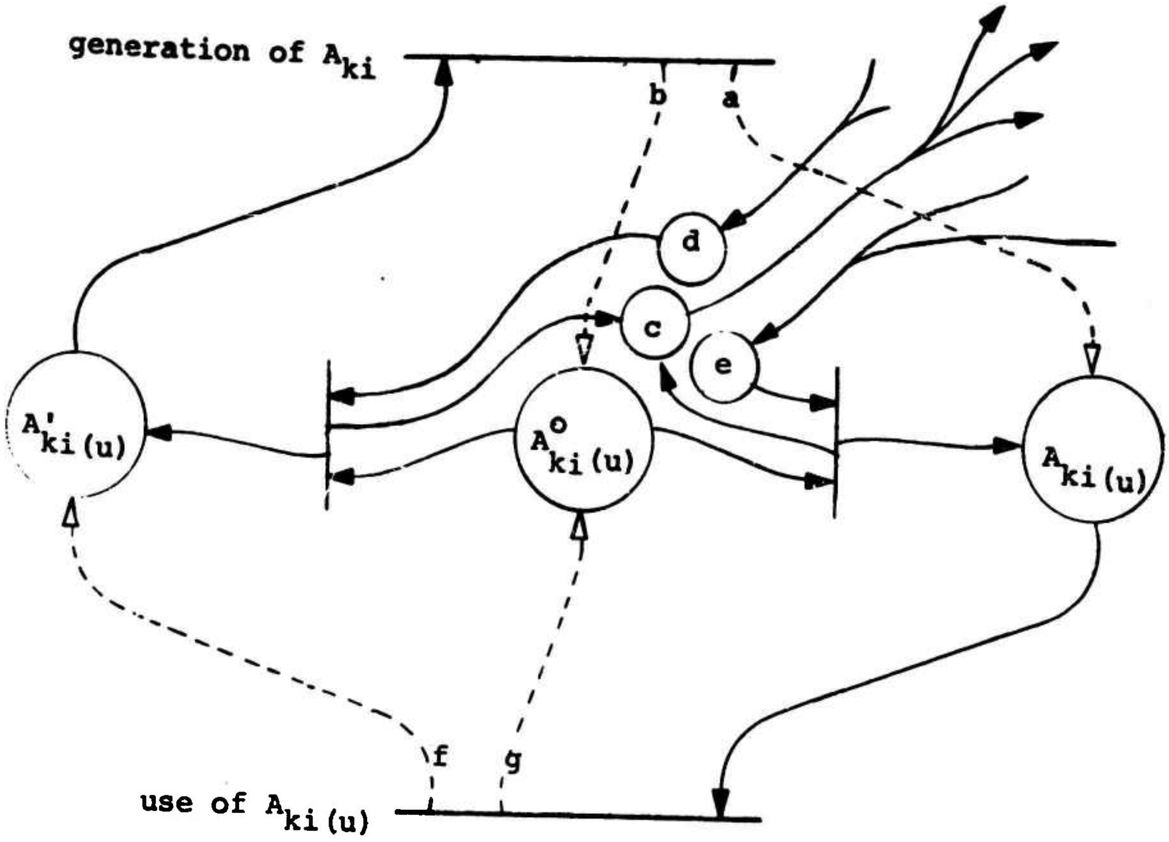
For each assignment in an algorithm we will produce one such operation representation. We can represent decisions (i.e., IF statements) similarly, except that we will represent the various possible outcomes or decision-resolutions explicitly as net conflict.



A decision has a variable as an operand just like any other operation. (Note that for each decision in an algorithm there will be in each p-graph of some variable in the algorithm a unique decision-node corresponding to that decision. Similarly, for each decision-resolution there will be a unique arc -- one of the output arcs of the decision-node -- corresponding to that decision-resolution.) Since we are aiming at a representation which explicitly exhibits data dependencies and since these data dependencies are determined by the interaction of control with variable-names, we will want, roughly speaking, to link decision results directly to variable-uses to generate ordering relations between operations. Therefore we will expand our previous representation of a variable-use from:



to:



(<u>k</u> stands for either <u>f</u> or <u>c</u> ; hence, <u>ki</u> is a subscript identifying the equivalence class of which the use is a member.

<u>u</u> is a subscript which unqiuely identifies the particular variable-use being represented.

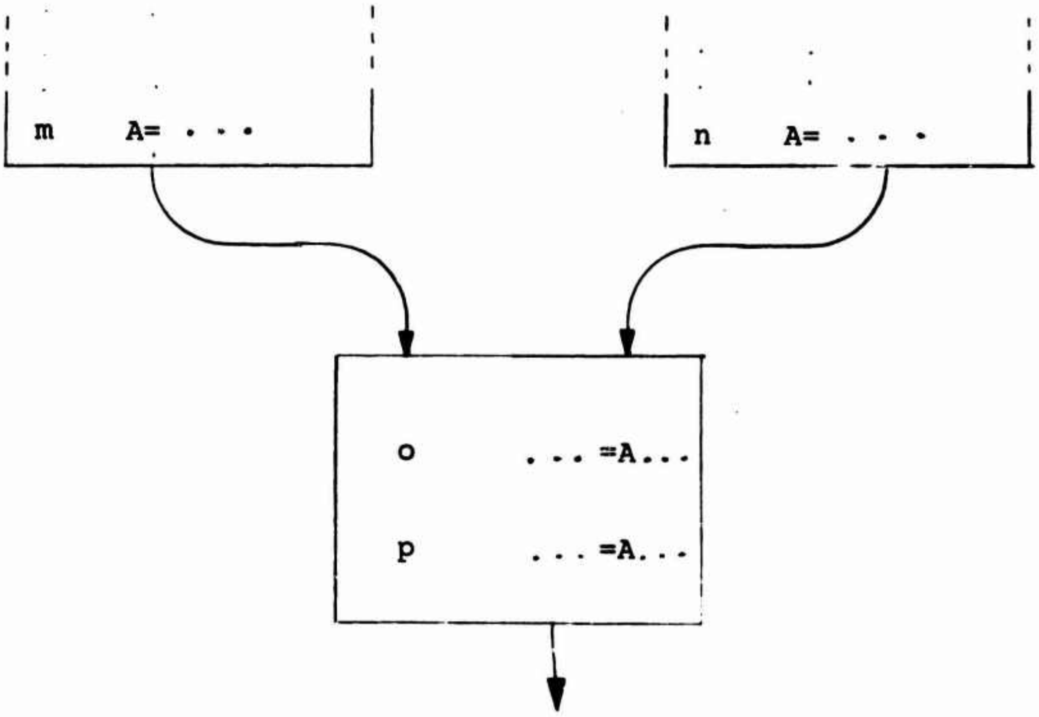
Arcs <u>a</u> and <u>b</u> are alternatives (exactly one is present in any given representation) as are arcs <u>f</u> and <u>g</u> .

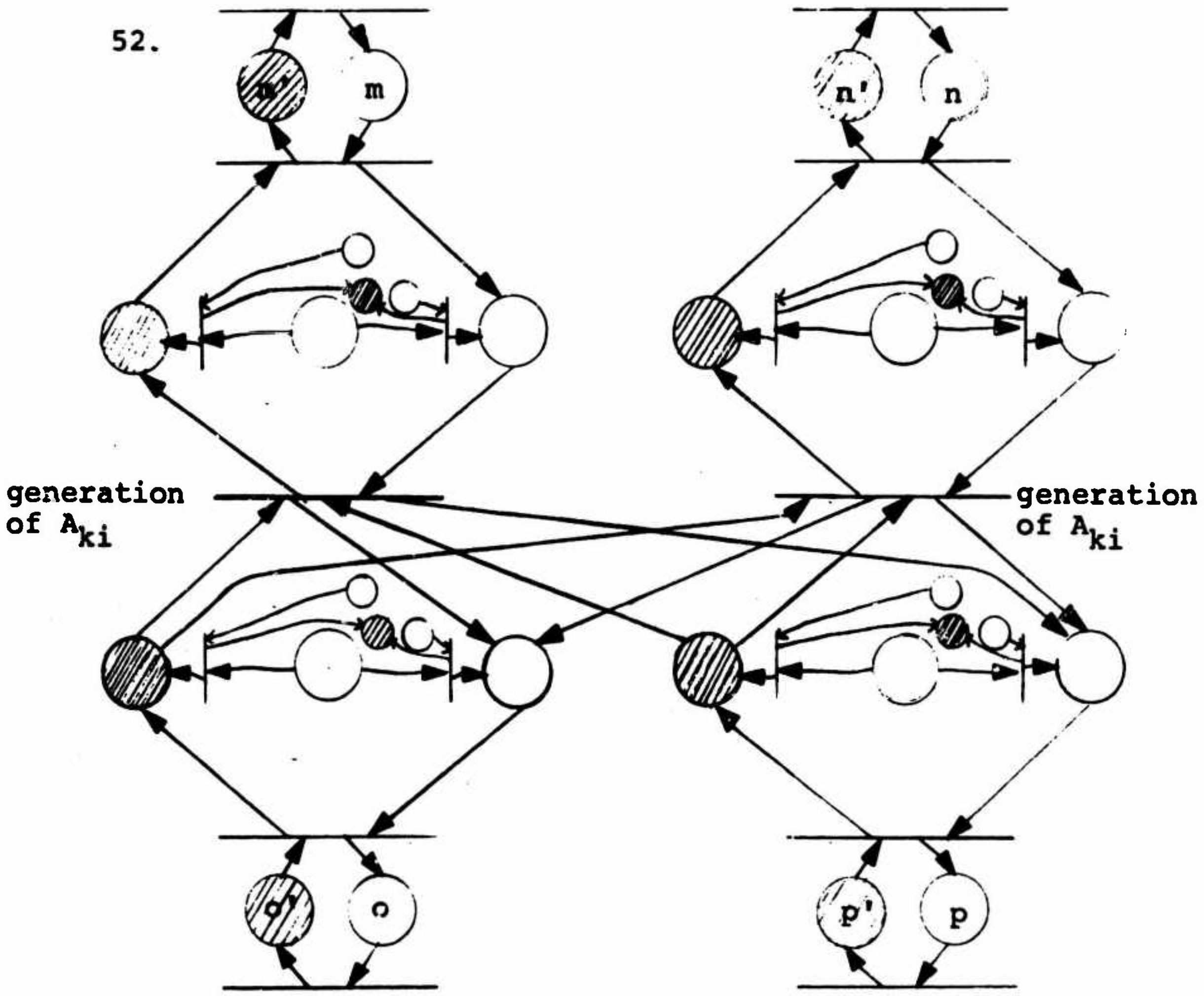
Places <u>d</u> and <u>e</u> represent decision results.)

Before giving formal rules for applying this schema, let us describe it in informal, approximate terms. $A_{ki(u)}$ has the same interpretation as in our previous schema: the current value of the variable is available for this use; it may not yet be changed to the next value. Similarly, $A'_{ki(u)}$ means: the current value has already been used and is no longer needed; it may be changed to the next value. $A^{o}_{ki(u)}$ represents a sort of limbo: the current value is available, but this use of it may or may not take place (before the generation of its next value), depending on the outcome of one or more decisions. Places c , d , and e are connected to transitions representing decision-resolutions. A decision-resolution which causes this use of A to take place (before the next generation of the equivalence class) has e as an output (and is called "an enabling event" for this variable-use) and c as an input. A decision-resolution which guarantees that this use of A will not take place (before the next generation of the equivalence class) has d as an output (and is called a "disabling event" for this variable-use) and c as an input. c means: the last decision result affecting this variable-use has "taken effect"; the next relevant decision result may be generated.

The transition labeled "use of $A_{ki(u)}$" represents the initiation of the operation in which this instance of A

is an operand. If the equivalence class A_{ki} is generated by an assignment, then the transition labeled "generation of A_{ki} " represents the completion of the operation which provides values for that equivalence class. All representations of variable-uses which are members of the same equivalence class will, of course, share the same generating transition and have different use transitions. If the equivalence class A_{ki} is generated by a merge of several A equivalence classes, we create a set of alternative generating events -- one for each equivalence class which participates in the merge. Each such generating event will consist of one transition which has as an "operand" a variable-use representation which is a member of one of the merging equivalence classes. Each of these alternative generating transitions will, of course, generate the entire equivalence class. For example:





¶ The arcs a and b are alternatives -- exactly one of the two is present in any given representation of a variable-use. If each time the equivalence class A_{ki} is generated, this use must take place at least once, then arc a is present and not arc b . If, after each generation of the equivalence class A_{ki} , this use may or may not take place, then arc b is present and not arc a . Similarly, arcs f and g are alternatives. If for each generation of the equivalence class this use may take place at most once, then arc f is present and not arc g . If for each generation of the equivalence class this use may take place more than once, then arc g is present and not arc $\underline{f}$.

Let us now restate these rules more precisely. For each uncircled node in the complete p-graph of $\underline{A}$ which is not a flow-node or a decision-node, we will produce a variable-use representation in accordance with the schema above and the following rules. Consider any such node $A_{ki(u)}$, which is a member of some equivalence class A_{ki} .

- If the equivalence class A_{ki} is generated by an assignment, then the generating transition of $A_{ki(u)}$ is the termination transition of the operation corresponding to the generating node of A_{ki} .
- If the equivalence class A_{ki} is generated by a merge, then there is a set of alternative generating transitions for $A_{ki(u)}$ -- one ccrresponding to each immediate predecessor node of the circled A_{ki} node.
- If the node $A_{ki(u)}$ does not have as an immediate successor a circled flow-node, then its use transition is the initiation transition of the operation associated with it.
- If the node $A_{ki(u)}$ has as an immediate successor a circled flow-node, then its use transition is one of the set of alternative

generating transitions for the equivalence class defined by the circled flow-node.

- If every path from the circled A_{ki} node to an exit-node of A_{ki} contains $A_{ki(u)}$, then the representation of $A_{ki(u)}$ contains arc $\underline{a}$ and not arc $\underline{b}$.

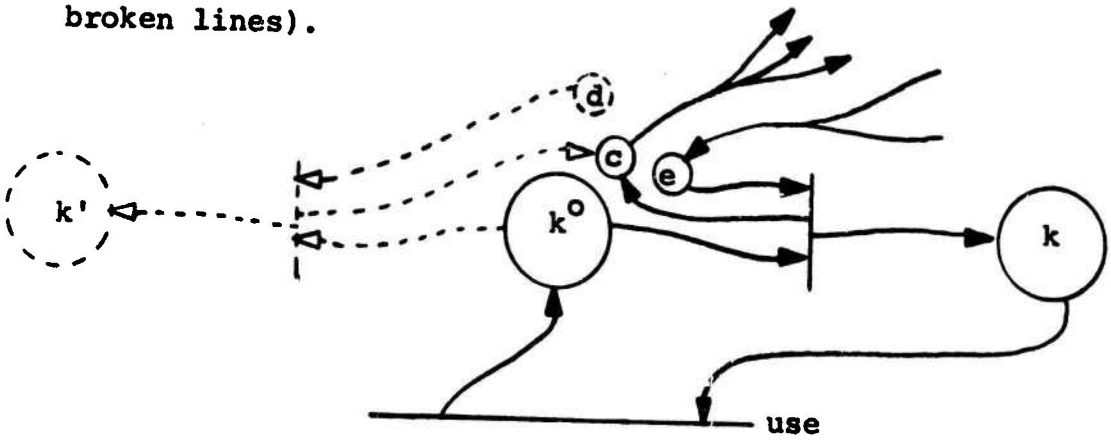
  Otherwise it contains arc $\underline{b}$ and not arc $\underline{a}$.
- If in the graph of A_{ki} there exists a circuit such that all nodes in the circuit are members of A_{ki} and such that $A_{ki(u)}$ is contained in the circuit, then the representation of $A_{ki(u)}$ contains arc $\underline{g}$ and not arc $\underline{f}$.

  Otherwise it contains arc $\underline{f}$ and not arc $\underline{g}$.
- For each decision-node $A_{ki(y)} \varepsilon A_{ki}$, such that there exists a path from $A_{ki(y)}$ to $A_{ki(u)}$ which is contained in A_{ki} , and such that there exists at least one path from $A_{ki(y)}$ to some exit-node of A_{ki} which does not contain $A_{ki(u)}$: Let P be the set of all paths p such that the first node of p is $A_{ki(y)}$ and the last node of p is an exit node of A_{ki} and such that the last node of p is the only node in p which is not a member of A_{ki} . Partition P into subsets P_1 , P_2 , ... , P_n according to the second node of each member path (i.e., according to the branch taken at the decision), so that each

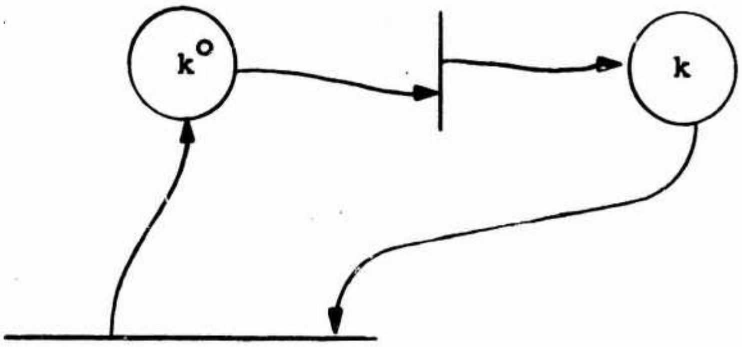
subset corresponds uniquely to a resolution of the decision.

- For each subset P_h such that all members of P_h contain $A_{ki(u)}$, let the decision-resolution transition corresponding to P_h have as an output place $\underline{e}$ (in the representation of $A_{ki(u)}$) and as an input place $\underline{c}$.
- For each subset P_j such that no member of P_j contains $A_{ki(u)}$, let the decision-resolution transition associated with P_j have place $\underline{d}$ as an output and place $\underline{c}$ as an input.

Constants are treated similarly. For each use of a constant, we produce a representation in accordance with the schema and rules for variable-uses. However, since there can be no generation event for a constant, part of the schema will be superfluous, (as indicated in the following figure by broken lines).

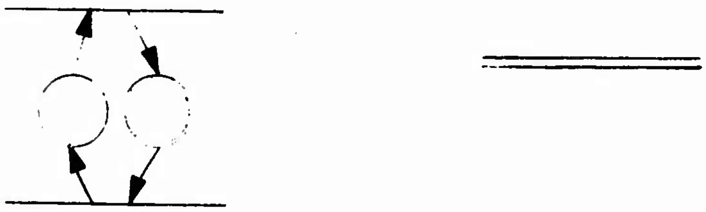


Furthermore, if place e is not an output of any transition -- i.e., if the constant-use in question occurs in every control history of the algorithm -- then we will eliminat: places e and c from the representation as well, so that the value is made available again after each use, independently of any other computation or decision. This would leave us with the following schema:

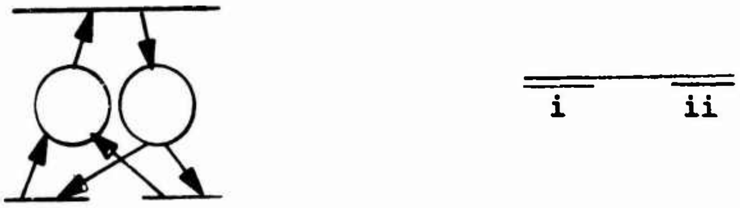


These representational schemata for variable-uses (and constant-uses) and decisions differ radically from conventional representations. A decision is no longer viewed as a point in a flow diagram at which control chooses one of several alternative paths and a decision-resolution simply as the choice of one of those alternative computational paths. Instead each decision-resolution has a set of results. Each of the results affects the status of some one variable-use, either enabling or disabling it -- i.e., each decision-resul determines either a forward or a backward data-dependency relation. One important aspect of this is that the various effects of a decision-resolution are given individual, explicit representation. Even more interesting, however, is the fact that this schema is free of the dualism of conventional representations: control and computation no longer have different ontological status; decision results and computational results alike are explicitly represented as conditions (or "sub-states" or "signals") in a partially ordered, cyclic system of such conditions.

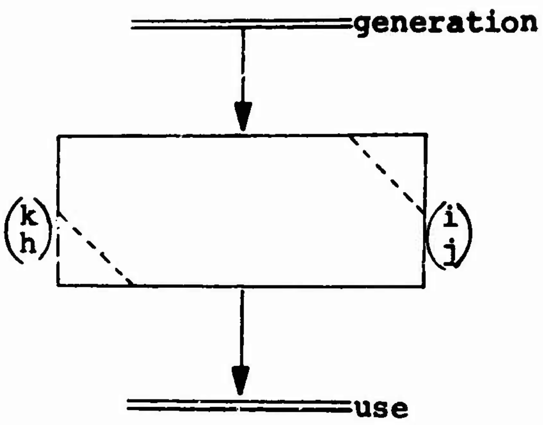
Having explained our representational schemata in detail, we will now replace them with more concise notational forms. We shall replace the operation schema with a double bar.



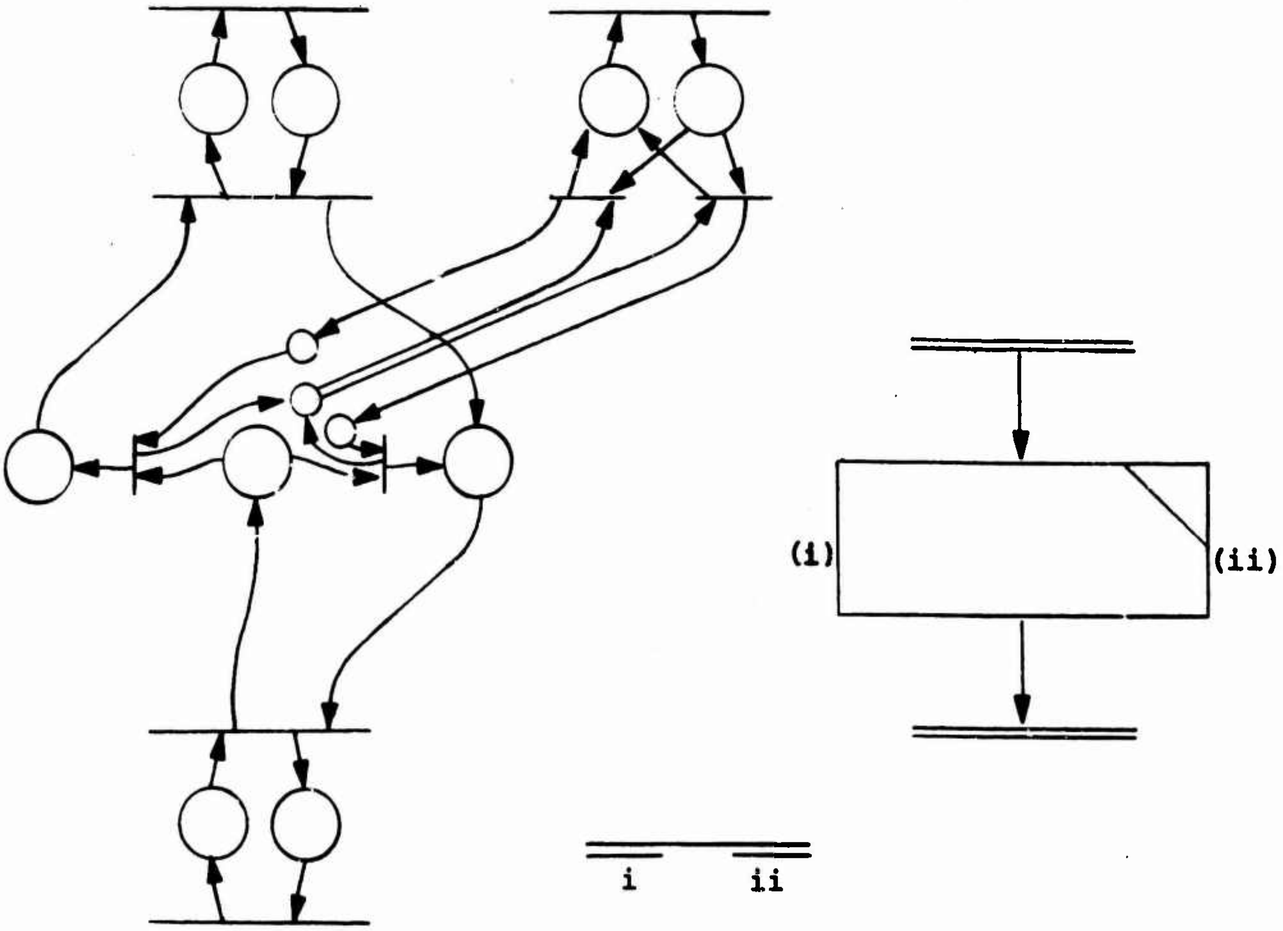
To model decisions, we shall break the lower bar to represent the various possible decision-resolutions. Furthermore, we shall name each decision-resolution in the algorithm uniquely.



We shall replace the variable-use model with a rectangle; an output arc will connect the generating transition to the variable-use model; an output arc will connect the variable-use model with its use transition. A diagonal in the upper right corner of the rectangle indicates the existence of arc a ; its absence indicates the existence of arc b . A diagonal in the lower left corner of the rectangle indicates the existence of arc f ; its absence indicates the existence of arc g . The names of all enabling events of the variable-use (i.e., inputs of place e) are listed along the right edge of the rectangle. The names of all disabling events (i.e., inputs of d) are listed along the left edge.



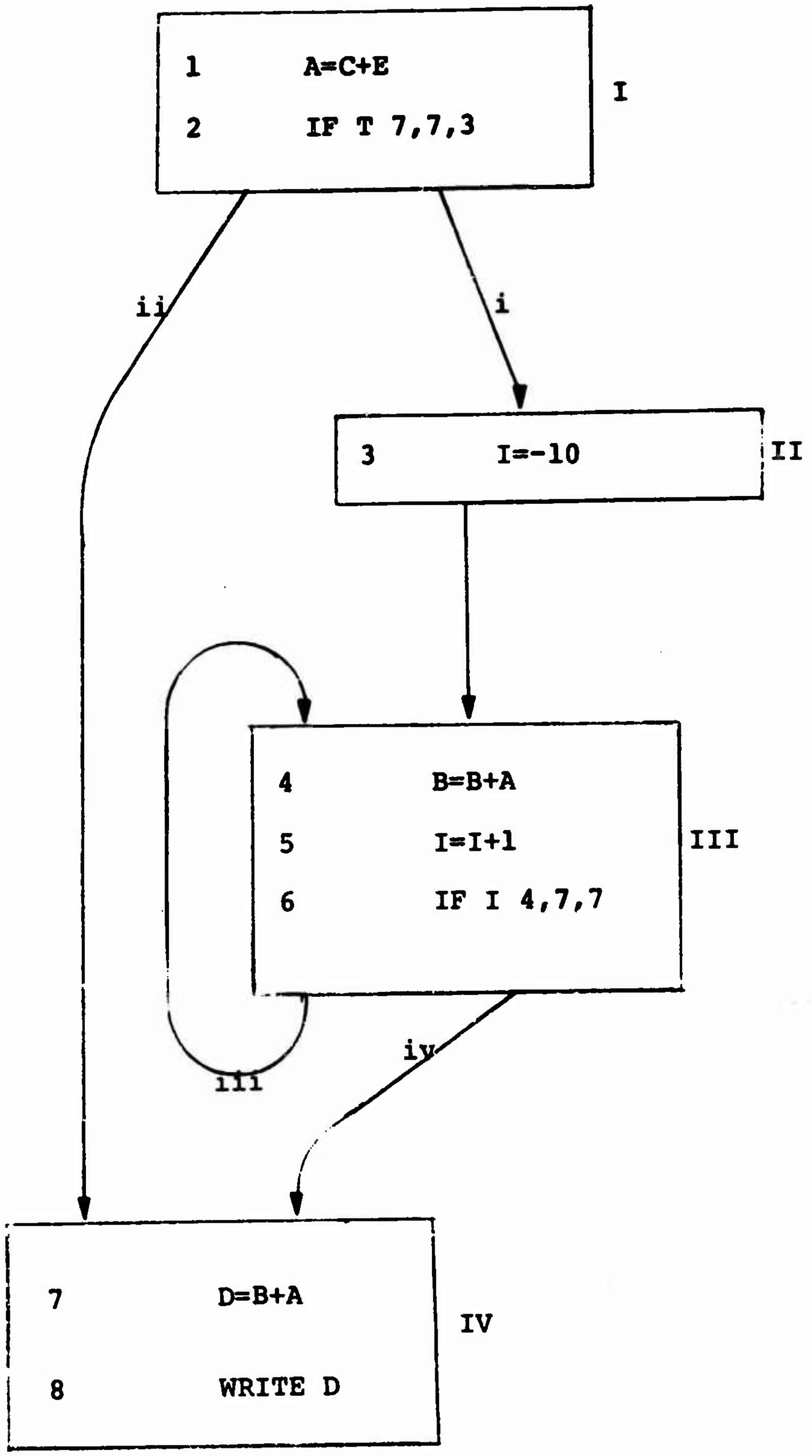
We can condense the following example accordingly.



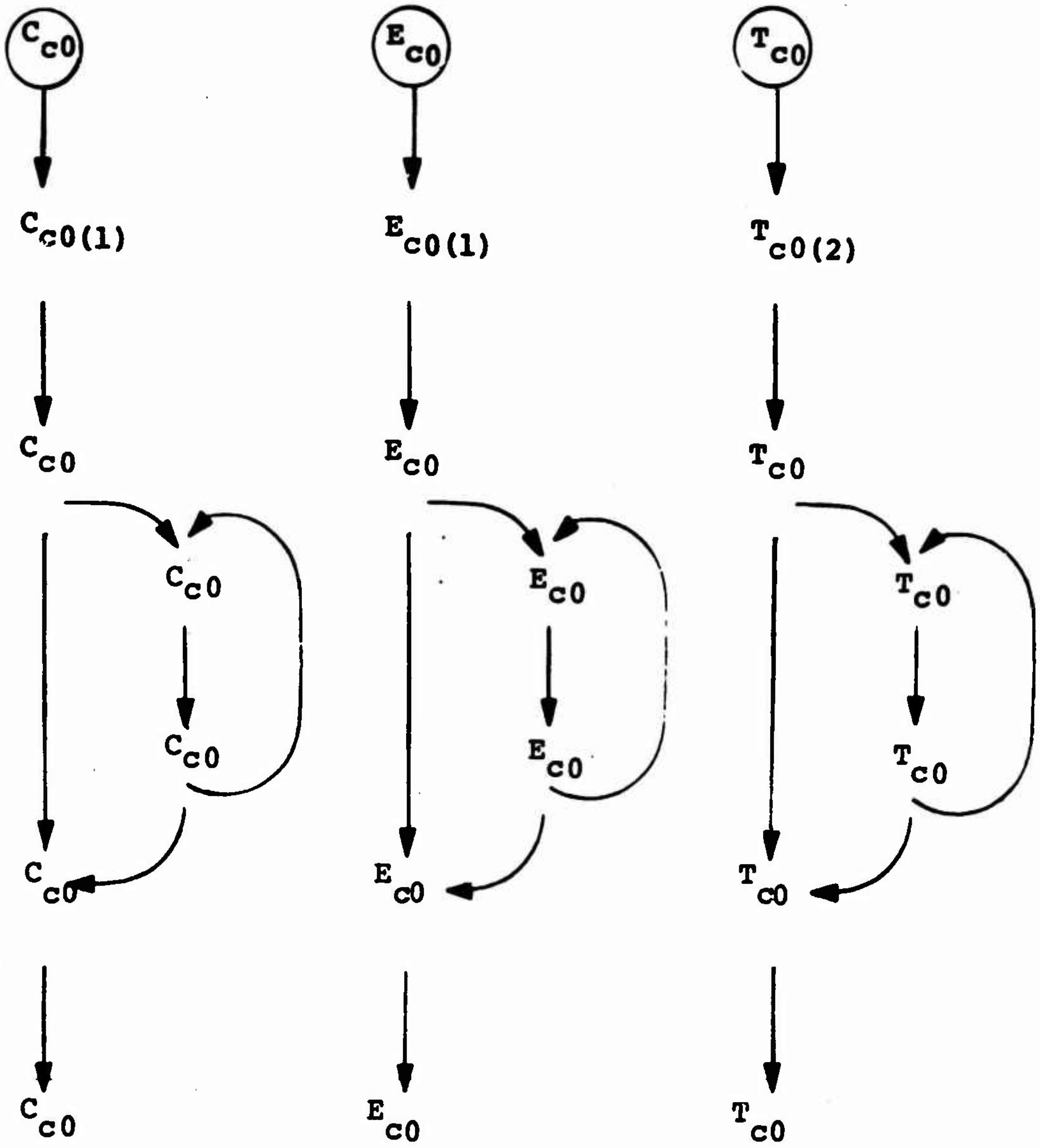
Before we can apply our representational procedures to an example, we must (for the time being at least) impose one further restriction: all decisions must be ordered. This is easily accomplished since every decision-resolution involves a commitment to a unique next decision. Therefore, for each decision $\underline{y}$ in the algorithm, we create a place which is input to the initiation transition of that decision; we can then make this place an output of every decision-resolution transition which has $\underline{y}$ as its immediate successor decision.

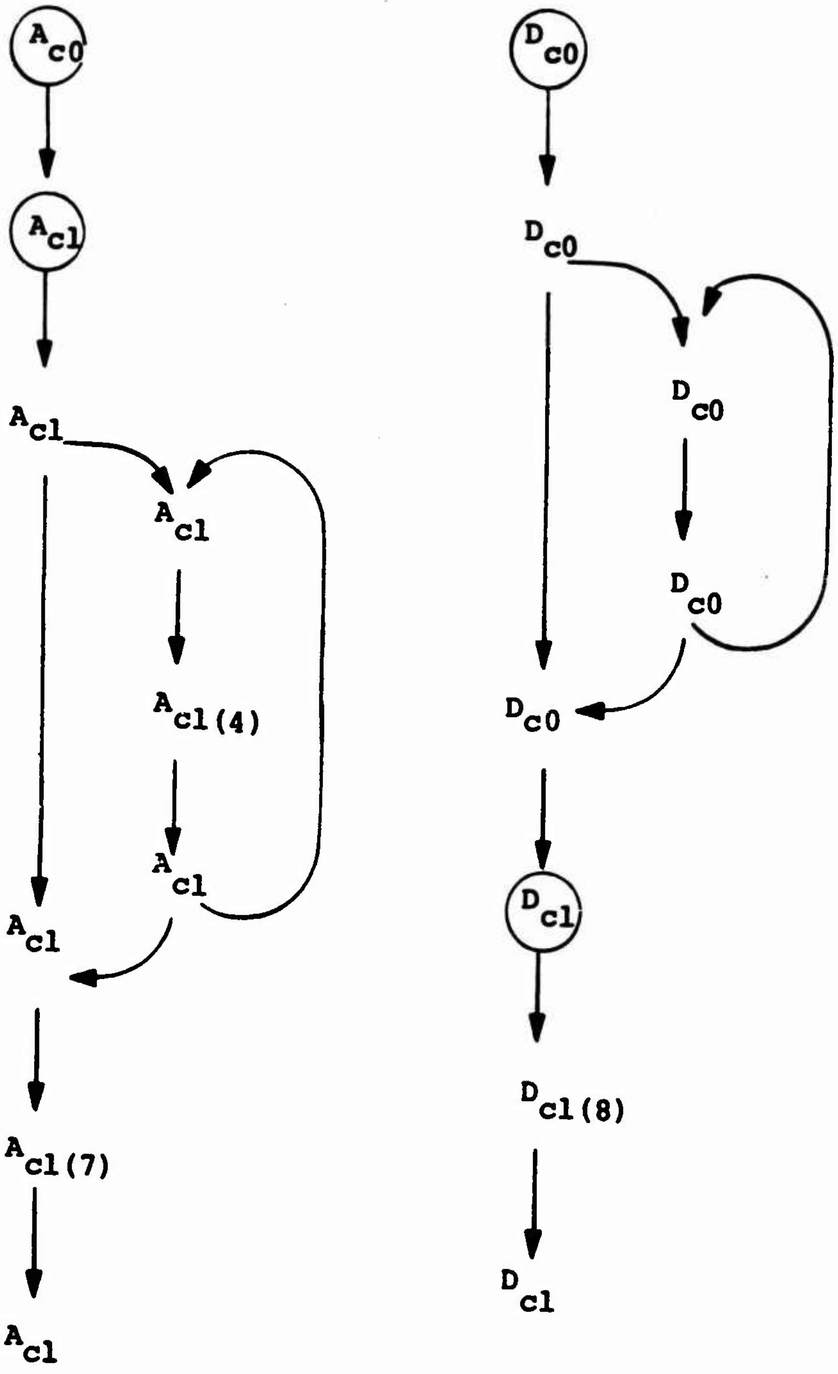
VIII. An Example of the Translation Procedure

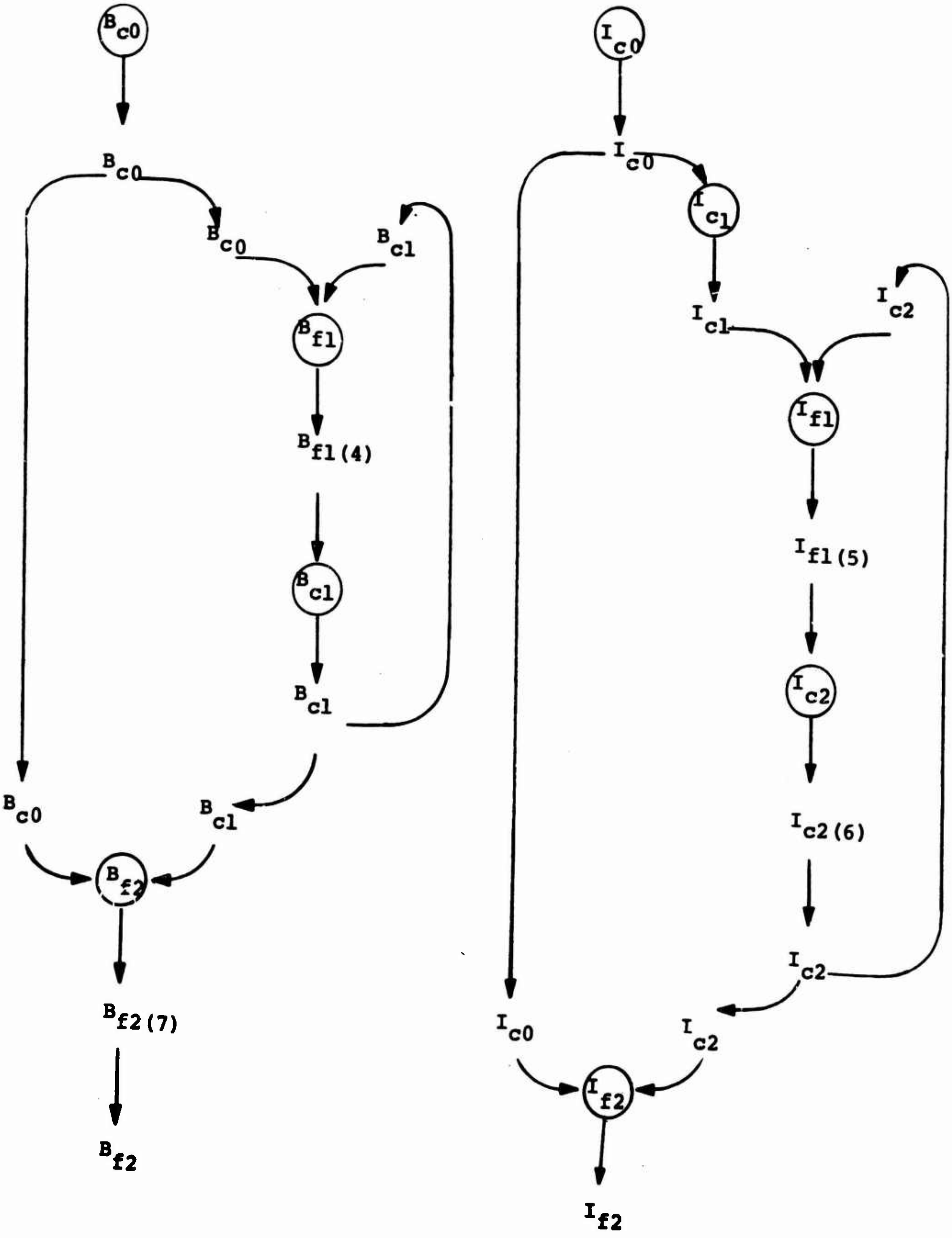
Let us now take the following algorithm segment as an example for translation into our representational form. For the sake of convenience and clarity we will number each statement and each decision-resolution.


1 A=C+E
2 IF T 7,7,3
I
ii
i
3 I=-10
II
4 B=B+A
5 I=I+1
6 IF I 4,7,7
III
iii
iv
7 D=B+A
8 WRITE D
IV

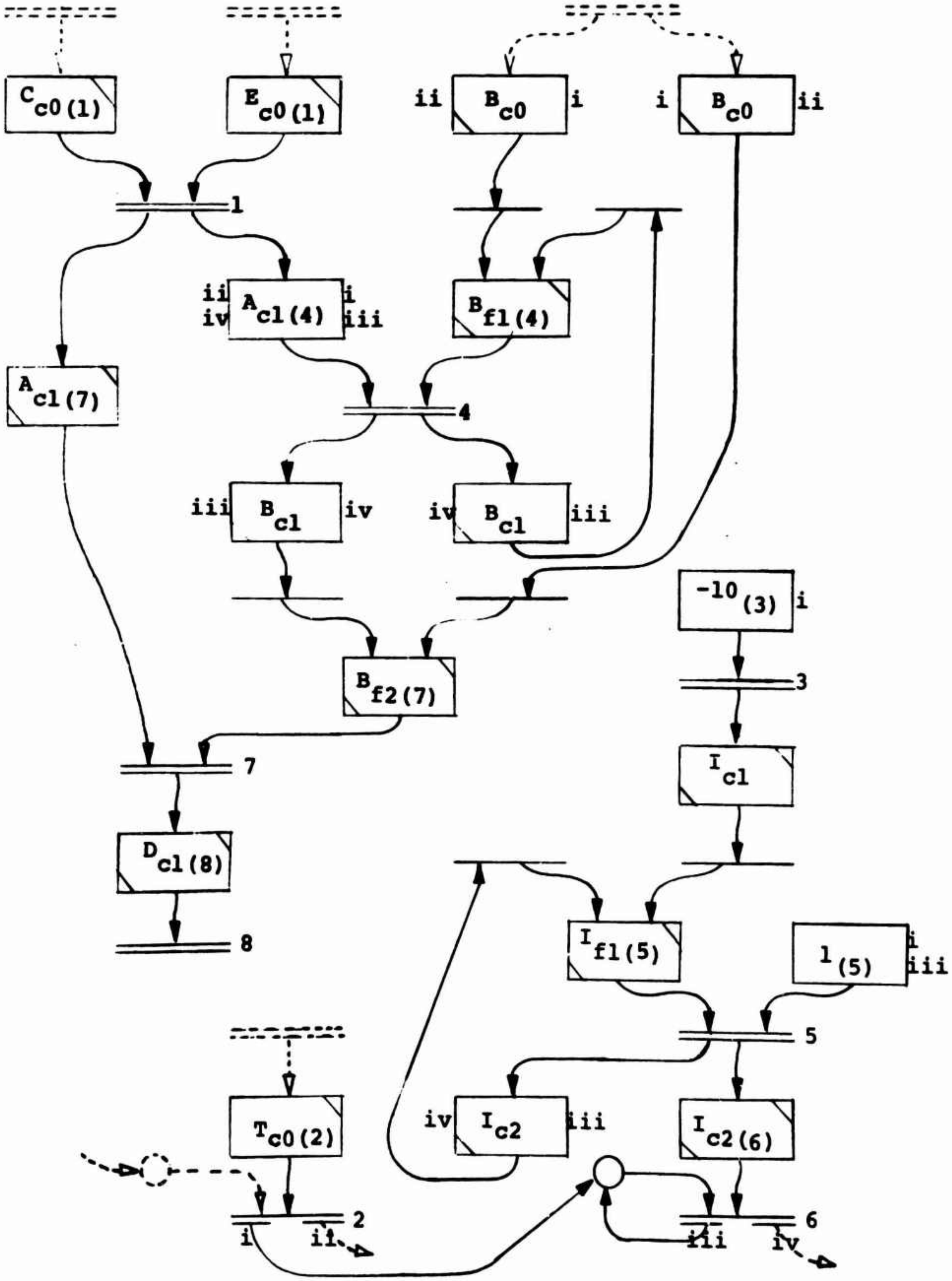
p-graphs for the algorithm




A_{c0}
A_{c1}
A_{c1}
A_{c1}
$A_{c1(4)}$
A_{c1}
A_{c1}
$A_{c1(7)}$
A_{c1}
D_{c0}
D_{c0}
D_{c0}
D_{c0}
D_{c0}
D_{c1}
$D_{c1(8)}$
D_{c1}


B_{c0}
B_{c0}
B_{c0}
B_{c1}
B_{f1}
$B_{f1(4)}$
B_{c1}
B_{c1}
B_{c0}
B_{c1}
B_{f2}
$B_{f2(7)}$
B_{f2}
I_{c0}
I_{c0}
I_{c1}
I_{c1}
I_{c2}
I_{f1}
$I_{f1(5)}$
I_{c2}
$I_{c2(6)}$
I_{c2}
I_{c0}
I_{c2}
I_{f2}
I_{f2}

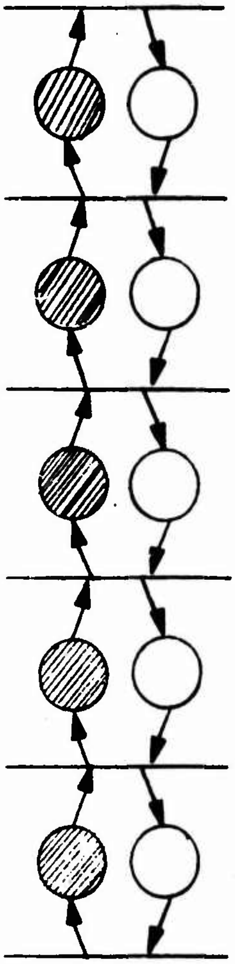
The Translated Representation of the Algorithm



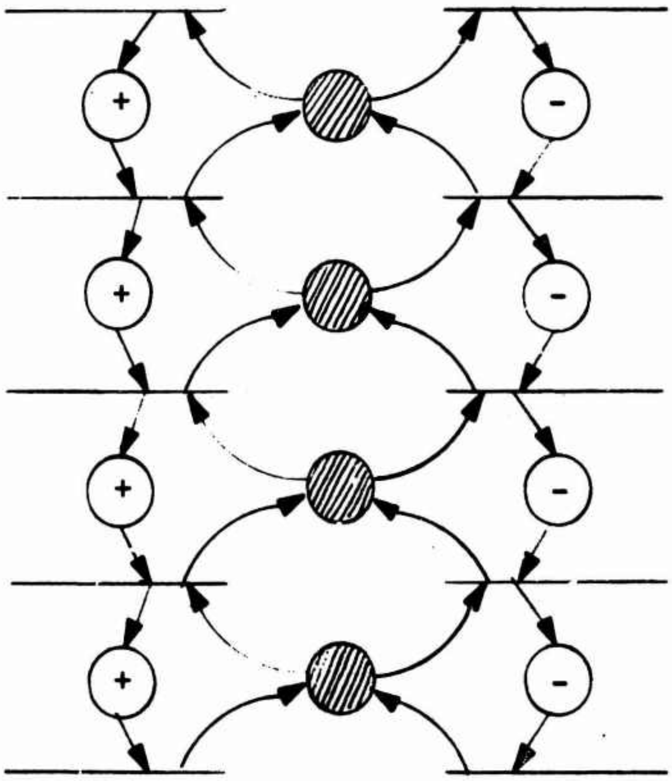
We now have a representation which expresses most of the obvious kinds of concurrency possible for the algorithm. It consists of a partial ordering of operations determined exclusively by the data dependencies (with the exception of the ordering of decisions). Control has been largely eliminated. Each decision interacts explicitly with each variable-use it affects.

IX. Pipelining

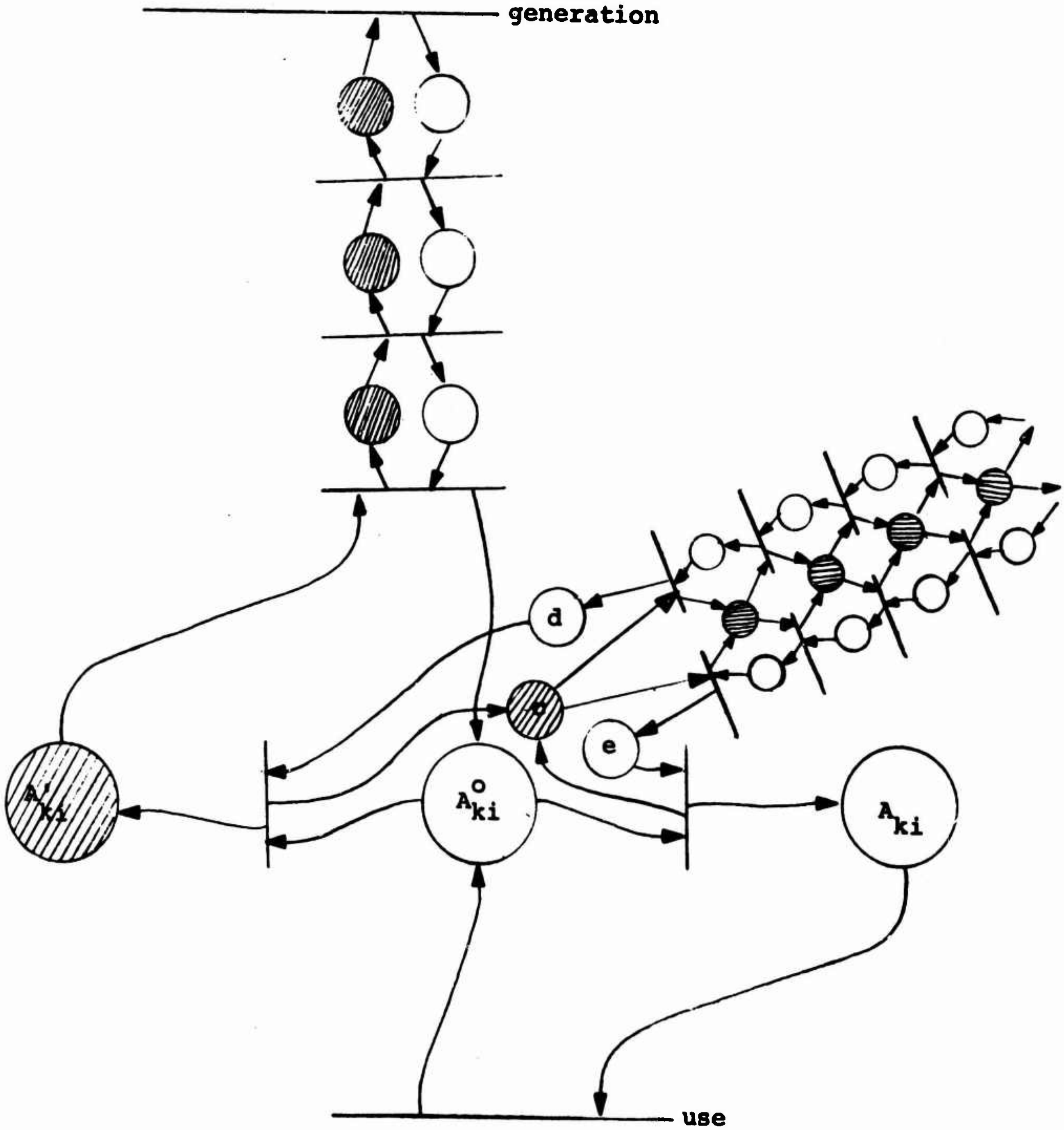
There is, in this representation, a certain amount of "play" between decisions and value availability: a value may be available for use in an operation before a commitment has been made to perform that operation; conversely, the commitment to perform an operation may be made before a necessary operand value has been generated. Because of the fact that algorithms contain cycles, it will be to our advantage to increase this freedom as much as possible. For example, we might consider a loop in which the control variable is computed independently from the other operations in the loop. If we could get several iterations ahead with the decisions, we could "wind up" the loop and achieve a pipeline effect. That is, ideally it might be possible to have all the operations in progress concurrently so that the throughput rate for the loop would be determined by the time required for the longest individual operation. To allow this kind of concurrency we will introduce a simple net structure which might be variously interpreted as a buffer, a stack, a queue, or a pipeline.



We have already used this structure to illustrate pipelining. We might also interpret each pair of places as representing a location in which a value or a signal may be stored. If the left place is full, that location is empty and may receive a value. If the right place is full, that location contains a value, which it may transmit to the next location (it is not possible for both places in a pair to be full -- ιr empty). We could then view this structure as a first-in-first-out stack. Signals are dropped in at the top and taken out at the bottom in order; the stack may hold as many values concurrently as it has place-pairs. Let us now suppose that there are two kinds of signals which may be placed into a stack and that we would like to distinguish between them explicitly. Furthermore, we will want to preserve the order in which they enter the stack. We can represent such a bi-valued stack as follows.



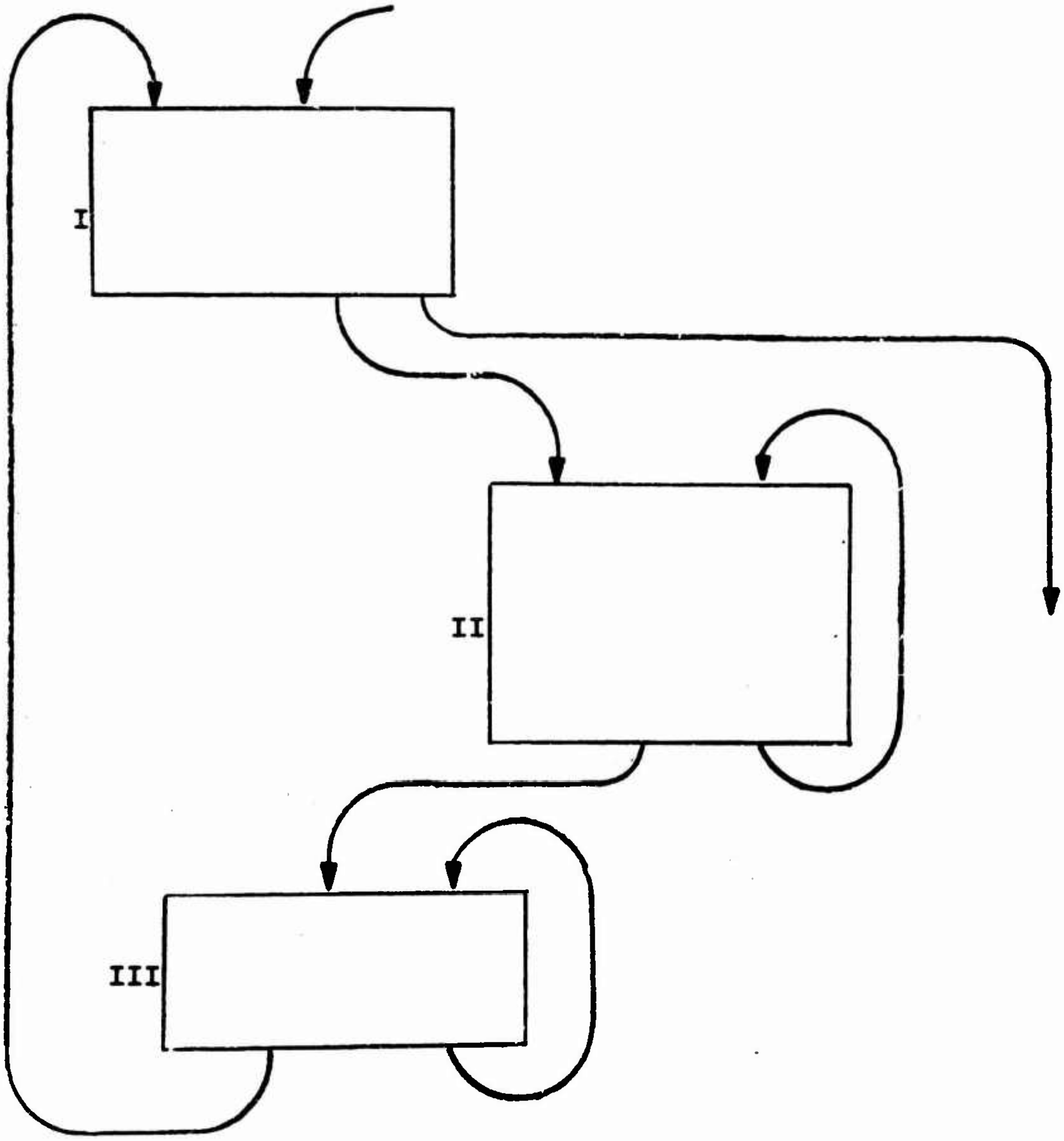
We can introduce such stacks into our representation of variable-uses and decision-results as follows.



In this fashion, we can create an arbitrarily high degree of freedom between decisions and computations. Furthermore, since the decision results affecting each variable use are given individual representation, this means that we may thereby increase the freedom between different computations.

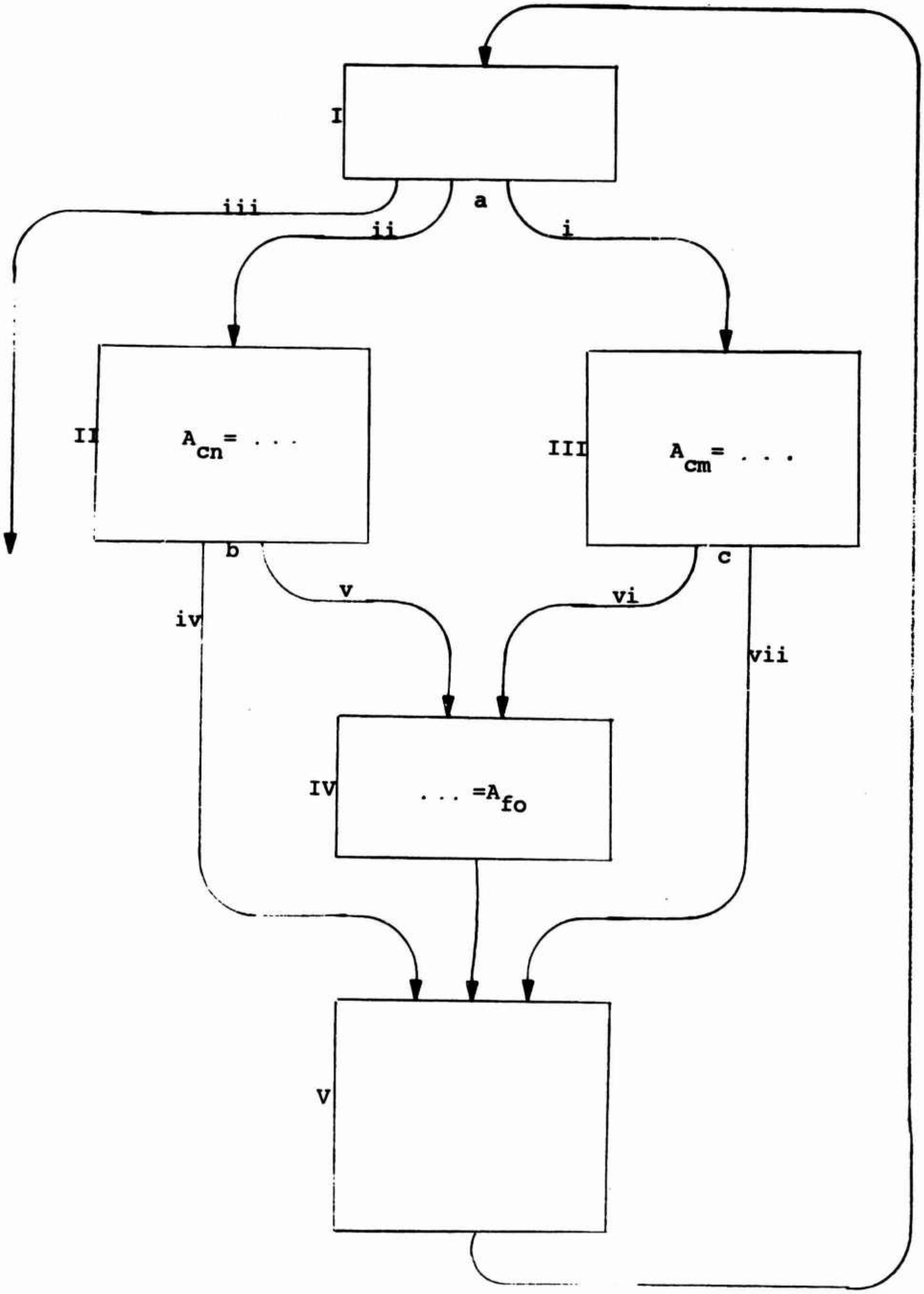
X. Control and Merges

We have considerably increased the power of our notation to represent potential concurrency, but our representation still contains arbitrary sequencing restrictions. The most obvious and serious of these is the ordering of decisions. Let us briefly consider two important implications of this restriction. First, we would like to be able to pipeline the algorithm as a whole so that it may concurrently process more than one set of inputs. As long as decisions are totally ordered, no significant amount of pipelining will be possible, since all decisions involved in the processing of one input set must clearly have been made before any decision involved in processing the next input set may be made. Secondly, let us consider the following example.



Suppose that loop II and loop III are unordered with respect to data-dependency (all values used in both loops might be generated in flowblock I, for example). There is, therefore, no data dependency constraint which prevents these two loops from "running" concurrently. As long as decisions are totally ordered, however, this possibility is excluded.

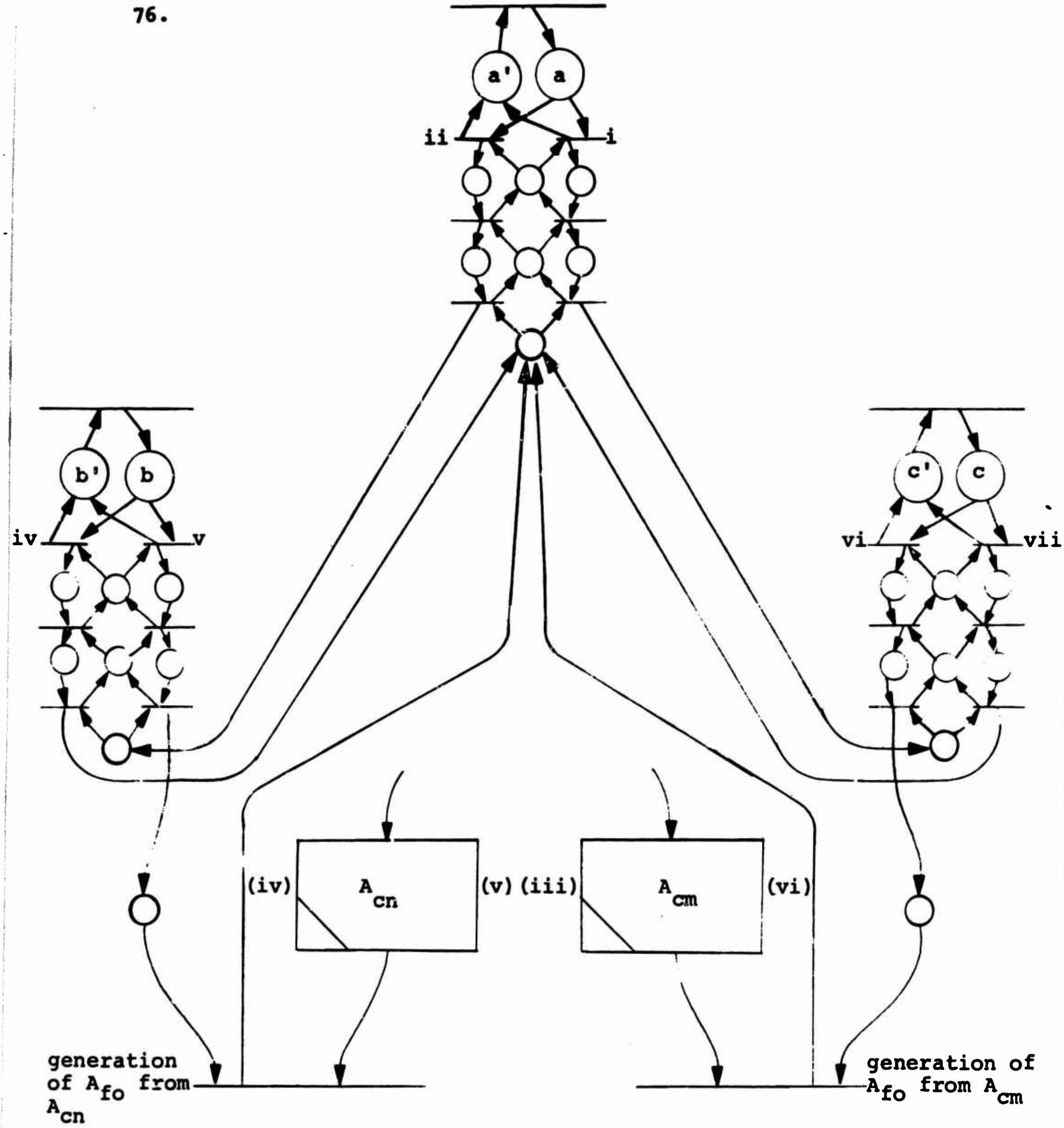
On the other hand, we cannot simply throw out the ordering of decisions altogether. To show why this straightforward solution is inadequate, we will try abandoning the ordering of decisions in the following example, in which we will be specifically concerned with the merge of the variable A at flowblock IV. We have named the decisions in this diagram a, b, and c, and we have named the decision-resolutions i, ii, iii, iv, v, vi, and vii .


I
a
iii
ii
i
II
$A_{cn}= . . .$
III
$A_{cm}= . . .$
b
c
v
vi
iv
vii
IV
$. . . =A_{fo}$
V

If we assume appropriate data-dependencies, the following set of events is possible: "Control" enters flowblock I and at decision a it chooses resolution ii . Let us assume that decision b is extremely time-consuming and that while this decision is being made, "control" (or "part of control", perhaps) skips ahead to flowblock VI and re-executes decision a -- this is, of course, permissible because decision a must be encountered again regardless of the outcome of decision b . Let us suppose that this time resolution i is chosen, and control enters flowblock III, where a value is generated for A_{cm} . Decision c is executed enabling A_{cm} to enter the merge and provide a value for A_{fo} . At this point decision b is finally completed; resolution iv is chosen enabling A_{cn} to provide a value for A_{fo} at the merge. Since the two "entries" into the merge occurred in the wrong order, however, any computations which use A_{fo} will have been rendered meaningless. Roughly speaking, wherever there is a merge (i.e., part-part matching), we must keep track of the logical priorities of the various claims which may be made on a representational "part". The different uses of such a "part" can only be distinguished by the order of their occurrence. Hence we will want to determine which decisions are critical in maintaining priorities among "entries" into a merge. <u>If we then order the effects of these decisions on the merge, we</u>

can allow the decisions themselves to take place in any order.

We can briefly outline a procedure for identifying the set of decisions which are critical for a given variable-merge. Take the p-graph of the variable in question and delete all circles. Circle the exit node and each of the immediate predecessor nodes to the merge node in question. Reverse the direction of every arc in the p-graph and apply Warshall's algorithm. This will cause the desired set of decision nodes to be circled (and only those nodes). We can then use this information to order entries into the merge. In the example above, decisions a , b , and c constitute the set of interesting decisions for the merge into A_{fo} . We can order the effects of those decisions as follows: (Note that the order in which the decisions themselves take place is not affected).


a'
a
ii
i
b'
b
iv
v
c'
c
vi
vii
(iv)
A_{cn}
(v)
(iii)
A_{cm}
(vi)
generation of A_{fo} from A_{cn}
generation of A_{fo} from A_{cm}

The desequencing of decisions may also lead to a similar problem with certain variable-uses, and a solution like that for merges is applicable.

XI. Proposed Extensions of the Representational Form

We have outlined procedures which make possible the translation of a sequentially defined algorithm into a powerful representation of highly concurrent execution of the algorithm. Roughly speaking, each operation may take place when (1) the necessary operand values are available, (2) enough decisions have been made to guarantee that the operation will be required, and (3) enough decisions have been made to guarantee that no logically prior claim can be made on the algorithmic parts involved. All sequencing has been stripped out except that which is given by data dependencies or by priorities for part use. In the process, control has been dismembered and the useful information which it carries has been broken down into individual ordering relations.

This is as far as we can carry the development of this representational apparatus in this discussion, but we would like to mention several possible extensions and applications. For example, we have already mentioned the fact that one arbitrary restriction imposed by the notion of control is that nothing may be executed which is not computationally necessary. However, it may prove more efficient to defer some decisions -- to pursue one or more alternative branches provisionally before the choice among them has been made. We now have a representation which exhibits explicitly which variable-uses are affected by a given decision. Therefore, we could mechanically build decision-deferral into our representation by moving enable/disable connections to other variable uses which are later in the chain of data-dependencies -- so long as we provide logical machinery to discard rejected values. Where two such alternative paths merged, furthermore, we could extend the decision deferral by "unzipping" the merge -- that is, by duplicating representational structures logically later than the merge. We might use the technique of duplication in another context as well: if we could identify computational bottlenecks, we might very profitably duplicate the structures at these bottlenecks. If we had statistical information about the relative frequency of different entry paths into a given merge, we might also implement another type of decision deferral: we could "open" the most probable entry to the merge on a provisional basis, even though the necessary decisions to determine priority of entry had not yet been made. Again, we would need logical machinery for discarding unwanted values. Several of the above possibilities involve duplication -- i.e., part-part matching in reverse. Because the data dependencies are exhibited explicitly we can also move in the opposite direction. We have already discussed one kind of part-part matching which is a standard optimization technique: elimination of redundant computations. We have accessible the information necessary for a global attack on this problem. Where two similar operations have operands generated by the same transitions (i.e., where the operands are members of the same equivalence classes), we can combine them. That is, we can replace the two operations with one operation which generates an equivalence class representing the union of the two equivalence classes generated by the replaced operations.

XII. Implications for Hardware Design

Finally, we would like to make several remarks about machine design. As the theoretical limits on the speed of computing components are approached, further increases in computing rates depend increasingly on our ability to build and use machines with highly parallel operating capabilities. Leaving aside the question of cost (which in any case can only be evaluated when we have the means to determine how effectively such equipment could be exploited), the principal problem in designing such computing equipment is not one of devising suitable physical components. The principal problem is rather the organization of physical components into a programmable system. Even the most straightforward digital computer is highly parallel in its operation in one sense -- its operation represents a very complex system of partially ordered events. It is simply that this system has been constructed in such a way that the subset of events interesting to us as users of the machine will occur sequentially (or very nearly -- even on the "programmable" level of machine behavior we can cope with a limited amount of concurrency). Digital computers are designed in this way so that sequentially defined algorithms may be mapped onto them. It is because of this that they _are_ programmable. Consequently any significant reorganization of hardware to exploit more fully the possibilities of concurrent operation must depend upon an appropriate conceptual reorganization of the representations of mathematical processes which we wish to perform.

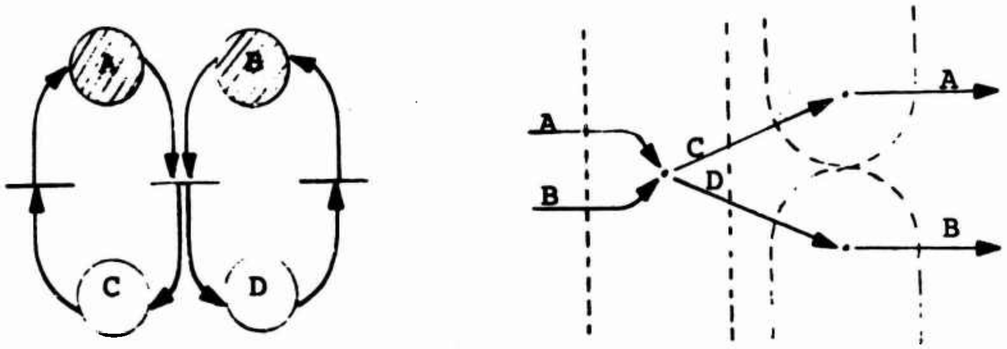
APPENDIX I

Petri Nets[1]

ɔrmally, a Petri net is a directed graph with two kinds of nodes: places, represented as circles; and transitions, represented as line segments. Each directed arc, represented as an arrow, connects one place with one transition. An arrow from a place to a transition means that the place is an input to the transition; an arrow from a transition to a place means that the place is an output of the transition. Every place in a net is an output of at least one transition and an input to at least one transition. No place may be both an input to and an output of the same transition. A place is capable of two states: full or empty. The state of a net is given by a list of all its full places. A transition may fire if and only if all of its inputs are full. When a transition fires, all of its inputs are emptied and all of its outputs are filled. If some place is input to two or more transitions, all of whose inputs are full, these transitions are in conflict. Only one of the transitions -- any one -- may fire in such a situation. (See Figures A, B, and C for examples of net diagrams. Figure B shows a net with conflict.)

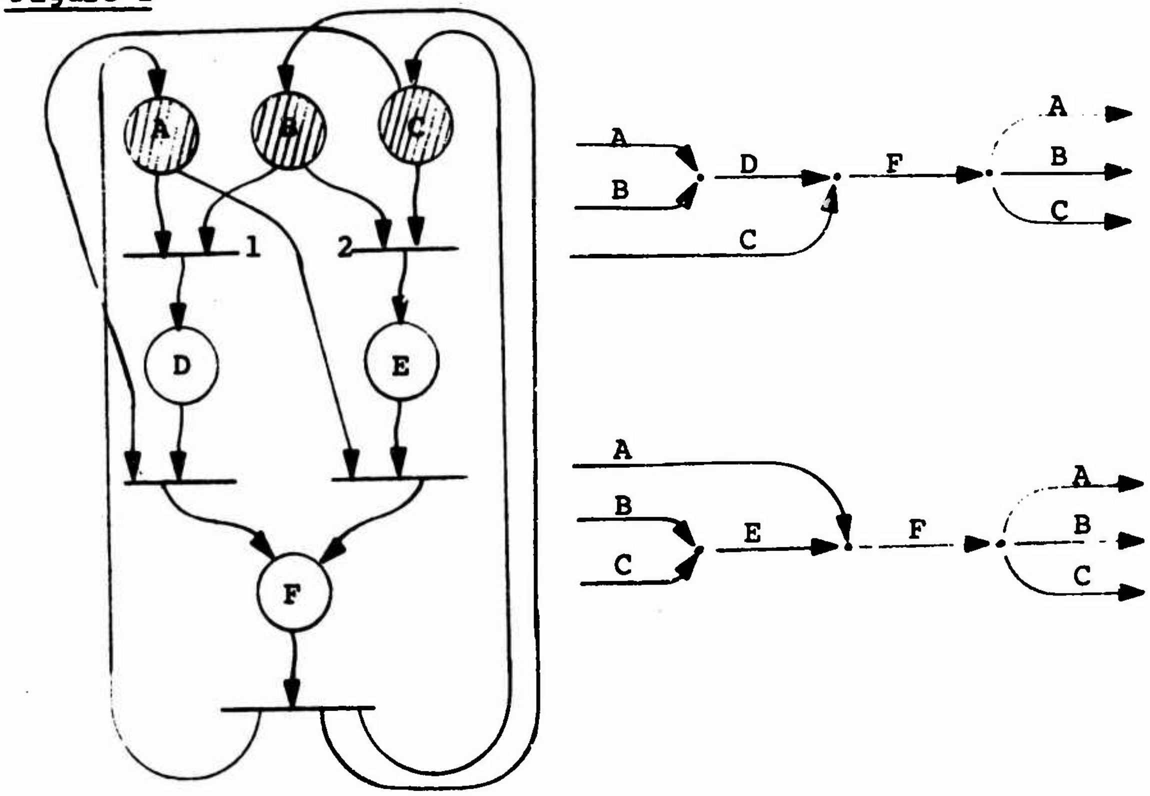
[1] For a comprehensive account of Petri nets we refer the reader to the "Final Report for the Information System Theory Project", RADC Contract # AF 30(602)-4211, by Dr. Anatol W. Holt et al.

Figure A



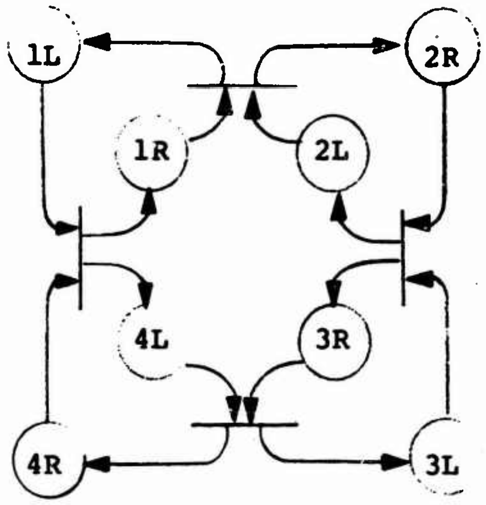
A net and an occurrence-graph representing its behavior. The shaded places are full. The broken lines represent time slices of the o-graph.

Figure B



A net with conflict and the o-cycles which constitute its basis. When A, B, and C are full, either transition 1 fires or transition 2 fires, but not both.

Figure C



1L : Ball 1 is moving
counter-clockwise.

1R : Ball 1 is moving
clockwise.

2L : Ball 2 is moving
counter-clockwise.

etc.

In using Petri nets to describe a system, each place is associated with a proposition about the system. By interpretation, when a place is full, the proposition associated with it is true. In other words, the condition described by a proposition holds in the system when the associated place is full. The state of a system described by a given state of its net is the conjunction of the propositions associated with the full places.[2] Thus a net diagram together with a suitable initial assignment

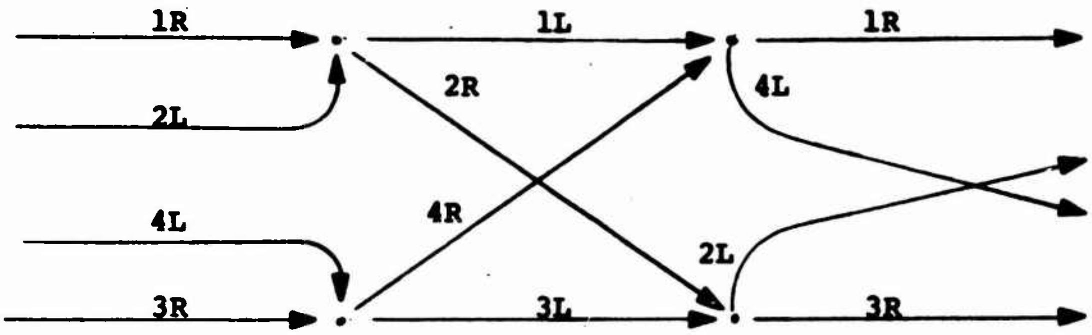
[2] It is perhaps misleading to speak of "system states" here since a net does not necessarily define a totally ordered sequence of states. (Formally, this is because some transitions may fire concurrently - that is, their firings are not temporally ordered.) In this respect, nets differ fundamentally from state machines.

of place states (corresponding to the conditions which hold in the system initially) makes possible a formal simulation of the behavior of the corresponding system. Note that it is the occupancy of places which is viewed as having duration. Transitions merely bound places; the firing of a transition is not viewed as time-consuming -- rather, it is a separation of distinct place occupancies. Hence, the propositions associated with places describe conditions involving time-consuming operations or states. Figure C, for example, is a net representation of four balls moving and colliding on a single-lane circular track. The propositions describing the system are all of the form: "ball n is moving clockwise (or counter-clockwise)".

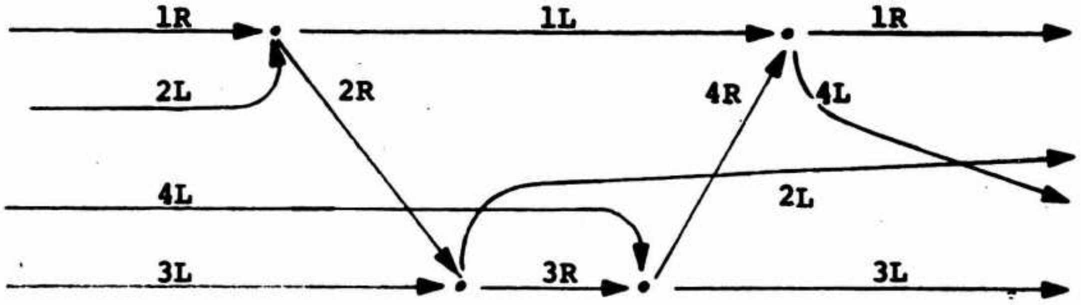
We may view an occurrence-graph, or o-graph, as a directed graph which represents a simulation history of some net. Formally, an o-graph consists of vertices, arcs, and labels associated with the arcs. Each label corresponds to some condition of the system being represented. (The words label and condition are therefore used interchangeably in this context.) Each arc represents an interval of place occupancy (or condition holding); the place (and hence the condition) is designated by the label associated with the arc. An inner vertex represents a transition firing and hence an occurrence in the system being represented. (The terms inner vertex and occurrence are accordingly used interchangeably.) Thus an occurrence may be described as follows: the conditions of the input arcs cease to hold (the input places become empty); the conditions of the output arcs begin to hold (the output places become full). (See Figures A, B, and D for examples of o-graphs.)

Two occurrences are said to be temporally ordered if and only if there is a path from one to the other; the former precedes the latter. Note that some occurrence pairs in an o-graph are temporally ordered while others are not. Occurrences which are not ordered are said to be concurrent. Similarly, two arcs are temporally ordered if and only if there is a path from one to the other; arcs which are not temporally ordered are concurrent. A time-slice is a maximal set of pairwise concurrent arcs. A time-slice represents a possible state of the net (and hence of the system) during the history which the o-graph describes. (See Figure A.)

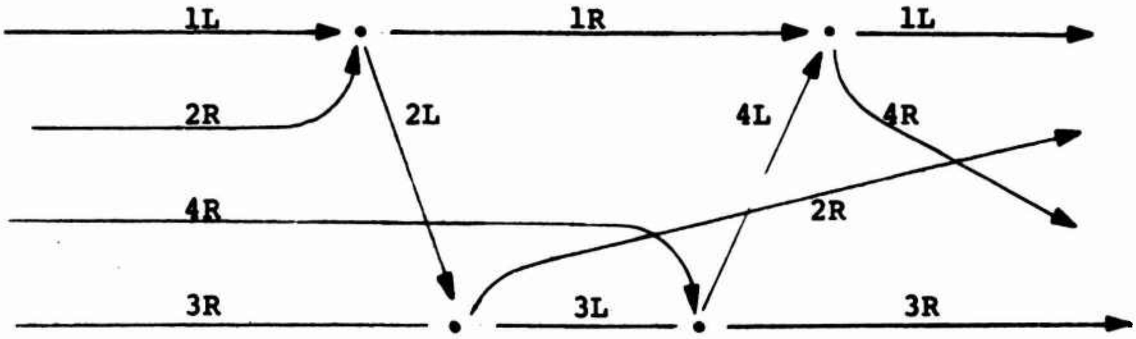
Figure D



(two balls moving clockwise and two counter-clockwise)



(three balls moving counter-clockwise and one clockwise)



(three balls moving clockwise and one counter-clockwise)

An o-graph may be decomposed at a time-slice. Two o-graphs may be composed if the terminal conditions of one are identical to the initial conditions of the other. An o-graph whose initial and terminal conditions are identical is termed an o-cycle. An o-graph formed by composing some number of copies of an o-cycle is termed a repetition stretch of the o-cycle. An o-cycle which cannot be decomposed into further o-cycles is termed an irreducible o-cycle. (The o-graphs shown in Figures A, B, and D are all irreducible o-cycles.) For every net together with a suitable assignment of place states, there is at least one basis, consisting of a finite set of irreducible o-cycles from which every possible simulation history may be generated by composition and decomposition. If the net contains no conflict, its basis consists of one irreducible o-cycle. Note that a given net diagram may be capable of several different disjoint behaviors given different initial place assignments. Figure D, for example, shows the bases for the three different behaviors of which the net in Figure C is capable.

APPENDIX II

Warshall's Algorithm

We start with a

definition: a p-graph is an ordered pair (G,N) where G is a finite, directed, single-source, single-sink graph, and N is any subset of the nodes of G which includes the source. For our purpose we may regard G as the flow graph of an algorithm where the unique entry and exit are the source and sink of the graph. N is precisely the set of initially circled nodes.

definition: a p-graph is complete if, for any node n of G either:

(i) $n \varepsilon N$; or

(ii) there exists a node $n^* \varepsilon N$ such that any path from any node in N to n includes n^* .

In terms of flow graphs, a graph is complete if every node has a unique circled ancestor, i.e., every use of a variable belongs to a unique equivalence class.

We now see that a solution to the naming problem is included in the solution to the problem of completing a p-graph. To further that solution we prove the key

theorem:

If (G,N_1) and (G,N_2) are both complete p-graphs, then $(G,N_1 \cap N_2)$ is a complete p-graph.

proof (Millstein):

$(G,N_1 \cap N_2)$ is trivially a p-graph.

Suppose it is not complete.

Then there exists $n \varepsilon G$ such that

(i) $n \not\varepsilon N_1 \cap N_2$; and

(ii) there exist $q_1, q_2 \varepsilon N_1 \cap N_2$, with paths p_1 , p_2 from q_1 , q_2 , respectively, to n such that p_1 , p_2 do not have a common point in $N_1 \cap N_2$.

case 1:

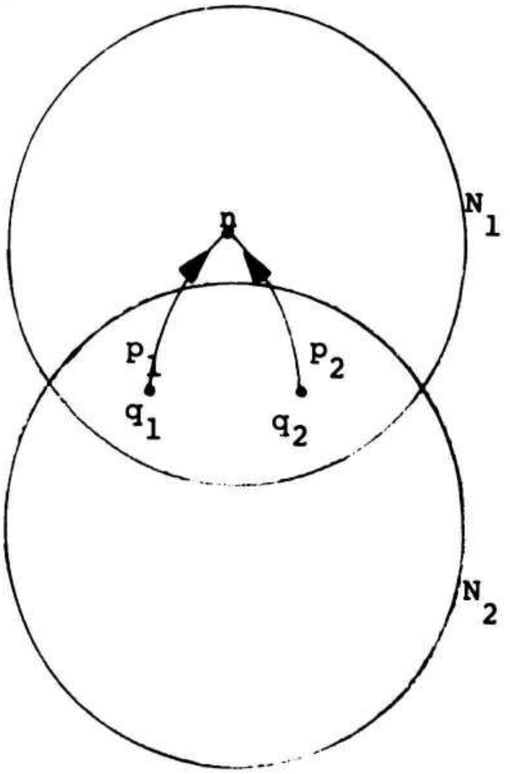
$n \; \varepsilon \; N_1 - (N_1 \cap N_2)$

Without loss of generality we choose

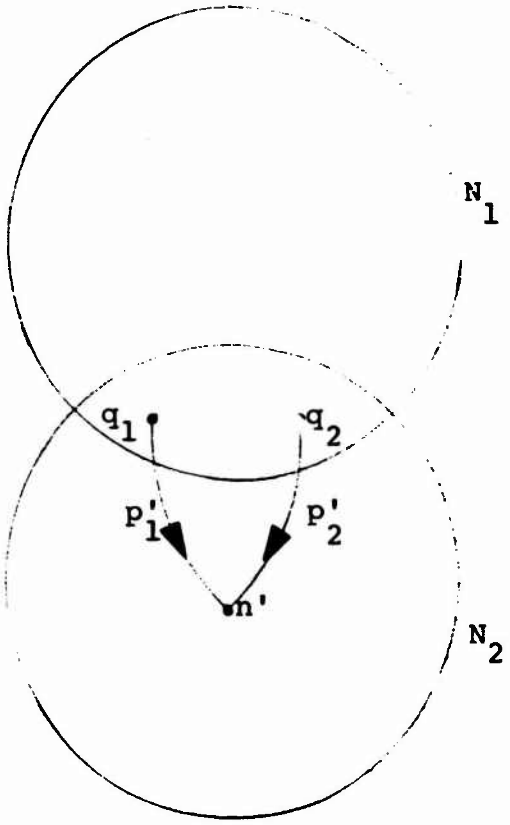
(a) p_1 , p_2 to be cycleless paths; and

(b) q_1 , q_2 to be the last points in $N_1 \cap N_2$ on paths p_1 , p_2 respectively; and

(c) n to be the first point in $N_1 - (N_1 \cap N_2)$ common to both paths. (Note: we use the finiteness in the definition of p-graphs in making these choices.)



Now (G,N_2) is complete. Also, $q_1,q_2 \varepsilon N_1 \cap N_2 \subseteq N_2$ and $n \not\in N_2$. Hence there exists $n' \varepsilon N_2 \ni p_1$, p_2 both pass through n' . Let p_1' , p_2' be the portions of p_1 , p_2 between q_1 , q_2 and n' .



Now (G,N_1) is complete. Also, $q_1,q_2 \varepsilon N_1 \cap N_2 \subseteq N_1$. $n' \varepsilon N_2$ and by assumption $n' \not\in N_1 \cap N_2$ (or else p_1 , p_2 have a common point in $N_1 \cap N_2$, contradicting (ii) above). Therefore, $n' \not\in N_1$ (and hence $n' \neq n$) .

Therefore there exists $n'' \varepsilon N_1$ such that p_1' , p_2' both pass through n'' .

Since $n' \neq n$ and p_1 , p_2 are cycleless paths, $n'' \neq n$.

Therefore n'' is a point in $N_1 - (N_1 \cap N_2)$ common to both p_1 , p_2 and $n'' \neq n$. This contradicts (c) above.

case 2:

$$n \varepsilon N_2 - (N_1 \cap N_2)$$

By symmetry of argument this case leads to a contradiction.

case 3:

$$n \varepsilon \complement(N_1 \cup N_2)$$

By a construction similar to case 1 this case reduces to case 1. Hence all three cases lead to a contradiction so (G,N_1, N_2) is complete.

Our main result is contained in the

corollary:

If M is an arbitrary subset of the nodes of G, there exists a unique minimal set N of nodes of G such that $M \subseteq N$ and (G,N) is a complete p-graph.

proof:

There exists at least one set with the required property: take all nodes of G. Moreover, since G is finite, there is only a finite number of sets with the required property. Therefore, we can take the intersection of all such sets and the result is the required minimal N.

We have shown that, given a p-graph, there exists a unique minimal completion of the p-graph. In this section, we give an algorithm for computing this completion.

We have defined the algorithm in a rather peculiar notation which requires some justification. The essential point is that the algorithm depends on cycling through the elements of a set, where the effect of processing an element may be to append other elements to the set.

If we attempt to express the algorithm in FORTRAN or ALGOL, we are forced to invent a data structure to represent the set: perhaps a linked list, perhaps a bit vector to indicate membership. In any event, we find ourselves making a decision about optimum representation, introducing new symbolic names (for the list head and pointers, or for the bit vector), and inventing cyclic controls of the loop-within-loop type which are more complex than the simple single quantification we started with.

In sum, FORTRAN or ALGOL representation of the algorithm is both complex enough to obscure the essential logical structure and quite arbitrary, in that a number of quite different-looking algorithms might be written without logical loss.

We have elected therefore to pay the price of an unfamiliar notation, in the hope that the very simple expression which results will disarm the reader's discomfort with a novel and not very well-defined language.

INITIAL CONDITIONS

We imagine as given:

1. D , a constant equal to the number of nodes.
2. VAL(I) , a vector where $1 \le I \le D$. VAL(I) = I if the Ith node of the given p-graph is circled; VAL(I) = 0 , otherwise.
3. S(I) , a family of sets, where $1 \le I \le D$. For any I , S(I) is the set of nodes which are immediate successors of the Ith node.

TERMINAL CONDITIONS

1. VAL(I) = I , if the Ith node of the completed p-graph is circled; otherwise VAL(I) = J , where the Jth node is the last circled ancestor on all paths to the Ith.
2. D and S(I) are unchanged.

VARIABLES

1. I , J , and Q are variables which assume integer values.
2. NOTYET is a variable whose value is a set of integers.

Algorithm:

```
COMPLETE

ALPHA $ Q←0.
        NOTYET ←{I|1≤I≤D}.
        (∀I|IεNOTYET∧VAL(I) ≠ 0)(BLEED(I) . NOTYET ← NOTYET-{I}.)
        If  Q ≠ 0 , (∀I|VAL(I) ≠ I)(VAL(I)← 0) .  GO TO ALPHA.
        EXIT.

BLEED (I)

        (∀J|JεS(I))(FLOW(I,J).) .
        EXIT.

FLOW (I,J)

        If VAL(J) = 0 , VAL(J) ← VAL(I) .  EXIT.
        If VAL(J) = VAL(I) , EXIT.
        If VAL(J) = J , EXIT.
        VAL(J) ← J .  Q ← 1 .  EXIT.
```

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Applied Data Research, Inc. - Corporate Research Center
450 Seventh Ave., New York, N.Y. 10001

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
N/A

3. REPORT TITLE
The Representation of Algorithms

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Final Report

5. AUTHOR(S) *(First name, middle initial, last name)*
Robert M. Shapiro
Harry Saint

6. REPORT DATE
September 1969

7a. TOTAL NO. OF PAGES
94

7b. NO. OF REFS
1

8a. CONTRACT OR GRANT NO.
F30602-69-C-0034

b. PROJECT NO.
4594

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
CA-6908-2331

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
RADC TR-69-313, Vol. II

10. DISTRIBUTION STATEMENT
This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTE

12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

The problem of representing mathematical processes is considered in the context of digital computer software and hardware.

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Multiprogramming						
Parallel processing						